TelePalestra – Ensino em *multicast*

Eduardo Filipe Amaral Soares
Mestrado Integrado de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2014

Orientador
Pedro Brandão, Professor Auxiliar, Faculdade de Ciências da Universidade do Porto

Coorientador
Rui Prior, Professor Auxiliar, Faculdade de Ciências da Universidade do Porto

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Dedico aos meus pais, irmã e avó que me ajudaram a ser o homem que sou hoje.
Um especial obrigado aos Doutores Pedro Brandão e Rui Prior que me transmitiram conhecimento e me deram oportunidades para fazer algo que me posso orgulhar.
Obrigado também a todos os meus amigos que me acompanharam e ajudam nos vários projetos pessoais e profissionais.

Abstract

This document tries to achieve a solution to medical data sharing in a school room or similar environment. The data can have any origin and is data type independent. But, uses as reference the teaching in the medical area, where are particular requirements about security, reliability and real-time.

The propose is making the sharing between devices from the most used operative systems (Android and Microsoft Windows), and constructing a solution that can run in all of them.

Some solutions are studied, between them: multi operative systems development, particular Web browsers and how they communicate. There are also explored and analysed protocols for reliable multicast.

A solution based in multicast is presented, giving the possibility for reliable or adapted for real-time sending of data. Given the requirements, all the data exchanged in the solution is safe and keys are exchange without the requirement for a remote server.

The work is presented as a software framework that can be used part of future applications, giving Web applications a service that they can connect and interact for creating, connecting and discover sharing sessions, and send or receive data to a channel of the connected session.

There is also included in the framework capabilities for building Java code that can be used for data collecting, processing and sending or receiving from the channels. This opens the possibility for the application in the Web Browser at runtime to call for their execution. Opening the application to go beyond the Web browser Application Programming Interface (API) limitations.

Resumo

Este documento tenta encontrar uma solução para partilha de dados em ambiente de sala de aula ou semelhante. Os dados podem ser de qualquer origem, sendo agnóstico do tipo. Mas, é tida como referência a área do ensino médico, em particular os requisitos de envio em tempo-real, segurança e fiabilidade dos dados.

O foco é habilitar a partilha entre dispositivos nos sistemas operativos mais usados (Android e Microsoft Windows), criando uma solução para aplicações poderem correr em todos eles.

Algumas soluções são estudadas, entre as quais: desenvolvimento multi-plataforma, em particular os navegadores de Internet e como estes conseguem comunicar para o exterior. São ainda explorados e estudados alguns protocolos para entrega de dados por *multicast* de forma fiável.

É desenvolvida uma solução baseada em *multicast* com possibilidade de entrega fiável e uma solução adaptada para tempo-real. É ainda tornada segura a solução, cifrando todos os dados de forma segura e com distribuição da chave sem necessidade de um servidor remoto.

A solução é apresentada na forma de uma *framework* base para desenvolvimento de aplicações Web podendo funcionar em multi-plataformas. A *framework* dá uma API para aplicações Web tomarem partido de um serviço para a distribuição de informação. Em particular este serviço permite criar e descobrir sessões de partilha de informação, ligar-se a uma sessão existente e receber ou enviar de canais de partilha de informação.

É ainda criada e incluída nesta *framework* uma solução para criação de blocos reutilizáveis para captura de dados e envio para os canais, em que há execução de código em Java de forma a remover limitações que houvesse ao correr a aplicação num navegador Web.

Conteúdo

Lista de Tabelas

Lista de Figuras

Lista de Blocos de Código

Capítulo 1

Introdução

Hoje em dia uma das coisas mais importantes na evolução da sociedade moderna é a partilha do conhecimento. Desde o início dos tempos a principal forma de partilha de conhecimento foi por forma oral [1].

Desde cedo foi notada a dificuldade de se manter toda a informação sem haver perdas ou acréscimos [2]. Por isso houve a necessidade de novas formas para transmissão de conhecimentos, inicialmente com os livros, evoluindo mais tarde para fotografias, vídeos e, nos dias que correm, com meios digitais (memórias não voláteis ou via Internet).

Com a facilidade de partilha nos meios digitais foram criadas novas ferramentas para converter do universo analógico para o digital. Um exemplo de uma destas ferramentas é um *scanner* [3] de imagens, permite ao utilizador converter papel para formato digital.

Esta evolução foi notória em todo o ensino. Não só hoje em dia grande parte dos manuais, enciclopédias, livros existem em formato digital, como também o conteúdo passou a ser mais rico e interativo. Este conteúdo faz-se acompanhar por vídeos ou até modelos 3D que podem ser explorados [4].

Uma das áreas que assiste a esta mudança com alguma relutância é a área médica. Por um lado os profissionais da área estão habituados ao meio analógico, e qualquer alteração de um sinal recolhido (por exemplo um eletrocardiograma ou uma auscultação), causado por a conversão para digital, pode ser perturbador. Por outro lado, existe a dificuldade de usar a capacidade de guardar e manipular informação por via digital para melhorar o ensino. Uma das maiores dificuldades que se encontra é na partilha e discussão destes sinais recolhidos. Meios tradicionais,

como Content Management System (CMS) destinados para a educação (como por exemplo o Moodle [5]), apresentam-se como soluções não ideais para o caso de uso.

O que este trabalho tenta alcançar é criar uma base para aplicações serem desenvolvidas para facilitar a partilha de informação em ambiente de ensino médico. Esta partilha pode ter diversas fontes e tipos de dados (que ficam ao cargo da aplicação recolher), alguns exemplos são:

- Eletrocardiograma: é a medição da variação do potencial elétrico criado pela atividade do coração. Um exemplo pode ser encontrado na figura 1.1 , existem diversos formatos de codificação para armazenamento do formato, uma discussão destas pode ser encontrada aqui [6];

- Ecografia: é uma técnica que através da reflexão do som pelas estruturas e órgãos permite uma visão sobre o interior do ser humano. Uma possibilidade para guardar a captura é o vídeo, um exemplo pode ser encontrado na figura 1.2.

- Auscultação: é uma técnica de escuta de sons internos do corpo (tipicamente do sistema respiratório e cardíaco), normalmente pelo uso de um estetoscópio como o da figura 1.3. Como a descrição indica, uma forma de guardar esta informação será como áudio sem perdas.

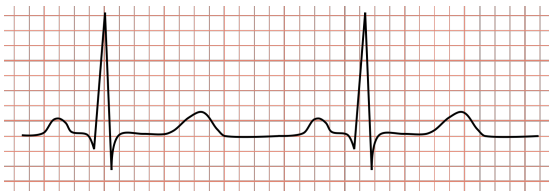


Figura 1.1: Exemplo de eletrocardiograma

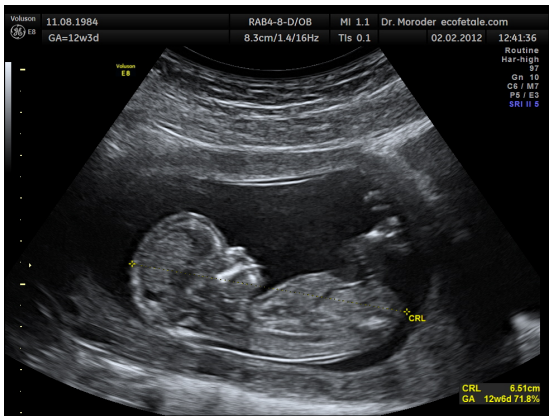


Figura 1.2: Exemplo de ecografia

Como descrito estes sinais podem ser recolhidos e codificados de várias formas, ou seja é necessária uma forma de transmissão que seja agnóstico da fonte e tipo de dados.

Figura 1.3: Exemplo de um estetoscópio

1.1 Ambiente da partilha de informação

O ambiente destinatário do trabalho desenvolvido centra-se num espaço fechado em que existe um sujeito principal responsável por gerir a sessão. Por exemplo um professor numa sala de aula, outro exemplo será uma palestra em que o palestrante a pode gerir. Assume-se que existe uma ligação local WiFi 802.11 gerida por um ponto de acesso e que todos os interessados têm um dispositivo com ligação a esta rede, que não tem necessariamente que ter ligação à Internet.

Os dados enviados podem ter sido previamente obtidos e armazenados, possivelmente com algum processamento ou compressão para reduzir a dimensão dos dados a enviar. Ou podem ser capturados no momento, em que uso de compressão pode não ser aconselhável pelo custo de tempo a processar.

Os dados a enviar podem ser transmitidos por a pessoa que gere a sessão ou terceiros, podendo até haver trocas de quem envia ou múltiplas pessoas a enviar. No exemplo prévio da sala de aula, pode ser o professor inicialmente a enviar uma captura em que posteriormente pede a um aluno para ser ele a enviar.

Devido à enorme variedade de dispositivos atualmente existentes no mercado decidiu-se focar nas principais plataformas de maior uso na atualidade, Android e Microsoft Windows [7, 8]. Em termos de Android, as estatísticas oficiais da Google, que detém a principal loja de aplicações na plataforma e é responsável pelo desenvolvimento do sistema operativo, indica que mais de 85% dos dispositivos [9] estejam na versão igual ou superior a *Ice Cream Sandwich* (4.0) . Foi

portanto escolhido focar o desenvolvimento para ser compatível com as versões mais recentes, pois é esperado que nos dispositivos futuros tenham a versão KitKat ou superior devido aos esforços feitos nesta versão para diminuir os requisitos de hardware necessários para a usar [10].

1.2 Objetivos que se pretendem alcançar

Tendo em conta as necessidades e o ambiente descrito delineou-se como principais objetivos para fornecer uma base para criar aplicações para partilhar informação:

1. **Fiabilidade na entrega**: toda a informação enviada deve ser recebida na totalidade e de forma ordenada por todos os recetores;

2. **Escalabilidade**: deve ser possível ter várias dezenas ou mesmo centena de clientes a receber a informação.

3. **Segurança**: dada a possibilidade de se tratarem de dados médicos é necessário que só as pessoas que podem ter acesso consigam obtê-los. Isto obriga a que todos os dados enviados sejam protegidos de terceiros, bem como cria a necessidade de autenticação dos clientes que vão receber os dados.

4. **Funcionar sem ligação à Internet**: dado o ambiente, uma ligação à Internet pode não ser viável. Depender em plataformas localizadas fora da rede local aumenta o custo de transmissão de dados, dado haver comunicação para o exterior para voltar à rede local. Pode haver ainda cenários em que seja inviável haver esta ligação, por exemplo numa pista de atletismo ou num campo de treino, locais ideais para recolha e análise de atividade física. Locais que por vezes não têm infraestrutura tecnológica, possíveis de colocar um ponto de acesso mas impossível de ligar à Internet.

5. **Em tempo-real**: por vezes em alguns casos de uso o envio e receção de dados em tempo real é necessária, para obter-se uma melhor discussão entre os intervenientes. Este ponto choca com o ponto 1, pois garantir que todos os dados são recebidos pode não ser possível num espaço de tempo suficientemente curto para obter a sensação de tempo-real. Um ponto a realçar é o facto de haver sempre uma distância temporal mínima entre o emissor captar os dados e estes chegaram aos recetores, e abaixo deste tempo não é possível baixar. O objetivo é o tempo que se demora não ser muito superior ao caso ideal.

6. **Multi-plataforma**: o resultado deste trabalho deve funcionar na maioria dos sistemas atuais no mercado. Focando-se nos que têm maior quantidade de utilizadores, mas deixando a possibilidade de implementações noutras plataformas.

1.3 Organização do trabalho

O presente documento começa por no capítulo 2 apresentar soluções existentes para alguns dos problemas que enfrenta e tenta atacar. Começando por o desenvolvimento em múltiplas plataformas (Android e Windows). As várias formas de comunicação de navegadores Web, dado estes serem uma das possibilidades de desenvolvimento multi-plataforma. Por fim refere e resume alguns dos principais pontos de alguns protocolos para *multicast* fiável.

No capítulo 3 são discutidas as possibilidades para construir uma solução e encontradas soluções para os vários problemas enfrentado. Sendo estas soluções implementadas e discutidos alguns detalhes de implementação e soluções dos problemas encontrados durante a implementação na secção 4. É ainda detalhado os problemas e soluções encontradas para executar em Android. Uma aplicação exemplo a usar a solução desenvolvida (a correr em Android e Windows) é demonstrada no capitulo 5.

Termina concluindo no capitulo 6 em que compara a solução desenhada e desenvolvida com os objetivos iniciais. Na secção 6.1 é apresentado os detalhes do que necessita de melhoramentos e o que não está implementado mas foi desenhado.

Capítulo 2

Estado da Arte

Dados os requisitos foi estudado o estado de arte no que toca a desenvolvimento para multi-plataformas de forma a suportar Android e Microsoft Windows (e até outros sistemas operativos para computadores, Mac OS e distribuições Linux). Foram também estudadas soluções para fiabilidade em *multicast* tendo em conta as possibilidades para o desenvolvimento e os requisitos.

2.1 Desenvolvimento Multi-plataforma

Para desenvolver para várias plataformas é preciso conhecer as limitações de cada plataforma. Em Android o Software Developement Kit (SDK) está apenas disponível na linguagem Java [11]. Apesar disto foram construídas ferramentas para suportar desenvolvimento em linguagens Web (HyperText Markup Language (HTML), Cascading Style Sheets (CSS) e Javacript) [12], algumas ferramentas notórias são o Apache Cordova [13] desenvolvido pela Apache Software Foundation e PhoneGap [14] que é uma versão do Apache Cordova com Application Programming Interface (API) extras fornecidas pela Adobe [15].

Apesar destas ferramentas de desenvolvimento aparentemente terem apenas aspetos positivos, vários opositores levantaram argumentos contra esta forma de desenvolvimento. Um dos principais pontos é o facto dos *browsers* embebidos nas aplicações terem problemas de desempenho, este é um problema atualmente a ser combatido e com melhoramentos ao longo das versões de Android. De momento já existem projetos que dão a possibilidade de usar o código fonte do chromium[1] e

[1]Chromium é um browser *open-source* disponível para várias plataformas. É também usado nas versões mais recentes de Android pela WebView, esta torna possível embeber páginas Web em aplicações. Mais detalhes do

usá-lo nas aplicações desenvolvidas, para assim ter disponível as últimas APIs e correções de erros [17], deixando assim de estar preso ao que a plataforma disponibiliza.

O outro dos principais argumentos apresentados é o facto do design para múltiplas plataformas resultar em baixa qualidade em todas, por não respeitar as diferenças entre elas [18]. Recentemente a Google lançou um novo conjunto de diretrizes de design no que toca à plataforma que são também válidos para a Web, colmatando assim as diferenças e tornando este ponto menos presente [19].

Já nos sistemas operativos para computadores existe uma maior possibilidade de linguagens de programação, sendo Java em geral suportada em todos. Há ainda a possibilidade de usar linguagens Web num *browser* ou usando o motor de um *browser* em conjunto com uma linguagem com APIs extras para colmatar as falhas que a API dos navegadores Web tenham. Um exemplo de um projeto que faz isto mesmo é o node-webkit, disponível em [20].

Assim sendo, o caminho comum encontrado para suportar múltiplas plataformas é o uso de tecnologias Web e Java para o que não for possível fazer nos *browsers*. Apesar de algumas modificações necessárias para as várias plataformas, o resultado final pode permitir a quem use a arquitetura construida ter de programar apenas nestas.

2.2 Navegador de Internet

Ao longo de muitos anos os *browsers* eram vistos como meros pedaços de software para mostrar texto formatado de acordo com a linguagem de formatação HTML [21]. Com a evolução da Internet e dos terminais que a consultavam apareceram necessidades para novas funcionalidades, que conduziram ao desenvolvimento de melhores capacidades de estilo com CSS [22] em 1996 e Javascript [23] em 1995, dando assim a capacidade para executar código na máquina que está a visualizar a página Web.

Com o crescimento da largura de banda disponível e a necessidade para combater o uso de *plugins* externos os fabricantes de *browsers* fizeram crescer as capacidades destes, através da definição da quinta versão de HTML. As capacidades criadas ou melhoradas são de várias áreas, passando por a recolha de vídeo e/ou áudio, capacidades para processar vídeo e áudio (ex. aplicação de filtros), geolocalização, comunicação entre o *browser* e o servidor Web (inicialmente

projeto podem ser encontrados em [16].

através de Hypertext Transfer Protocol (HTTP), posteriormente usando novos protocolos como Websockets), comunicação direta entre dois *browsers*, entre outros [24].

Para a biblioteca a construir de acordo com os objetivos definidos no capítulo 1, valia a pena analisar com detalhe as várias API disponíveis nos *browsers* que permitam comunicação de dados, seja para um servidor seja diretamente para outro *browser*.

2.2.1 XMLHttpRequest

XMLHttpRequest [25] é uma API criada para permitir fazer pedidos a um servidor de forma assíncrona, permitindo assim enviar ou receber dados de um servidor web sem necessidade de recarregar a página completa, implementado o modelo Asynchronous JavaScript and XML (AJAX). Apesar do nome sugerir a obrigatoriedade da utilização de Extensible Markup Language (XML) este é facultativo e o conteúdo pode ser codificado em qualquer formato.

2.2.2 Server-Sent Events (SSE)

Apesar de AJAX dar uma enorme número de capacidades à Web que até então não eram possíveis, este não permitia ser o servidor a iniciar uma comunicação com o *browser* quando necessário (ex. para comunicar algum evento ou a alteração de um estado). Soluções baseadas em AJAX foram criadas, a destacar entre elas o Polling, Long Polling (difere do anterior pelo facto de neste estar sempre um pedido pendente no servidor ao qual ele responde quando tem algo a enviar, enquanto no primeiro ele faz um pedido para obter dados pendentes regularmente) e Comet [26, 27].

Estas soluções têm um grave problema, o *overhead* do pedido HTTP. Em soluções em que se pretendam ligações permanentes este custem tem de ser tido em conta. Para minimizar este e outros problemas foi criada uma melhor solução chamada de Server-Sent Events (SSE), estandardizada por o World Wide Web Consortium (W3C) [28].

Através desta API no código é possível invocar no *browser* um pedido para ficar à escuta de notificações do servidor, o *browser* faz um pedido HTTP ao servidor. Pedido esse que potencialmente não termina, criando uma ligação permanente entre os intervenientes. Assim, quando o servidor quer enviar uma notificação ao cliente simplesmente escreve na ligação aberta de acordo com o formato apresentado no bloco 2.1.

```
[ID:numero de sequência]
data: {dados}
[data:{dados}]
```

Bloco de Código 2.1: Formato dos eventos enviados por o SSE

2.2.3 WebRTC

O WebRTC apareceu como um projeto com o objetivo de dar capacidades de comunicação em tempo real entre *browsers* [29]. Atualmente a API foi standardizada pelo W3C [30] e os protocolos que são usados para comunicação foram standardizados por o Internet Engineering Task Force (IETF) [31].

Esta API permite a dois *browsers* comunicarem entre si vídeo, áudio ou dados sem necessidade de um servidor pelo meio a coordenar e a encaminhar os dados de um cliente para o outro. Além da comunicação, o WebRTC inclui ainda codificação de vídeo e áudio para permitir a compressão dos conteúdos e adaptação à largura de banda disponível, conseguindo assim o envio com melhor qualidade de experiência para o utilizador. Os *codecs* obrigatórios ainda não foram definidos no standard, mas os recomendados a passar a ser obrigatórios na versão final podem ser encontrados em [32, 33]. Apenas existem *codecs* de áudio obrigatórios devido a questões de patentes sobre codecs de vídeo considerados [34].

A negociação da ligação e transferência de dados entre os *browsers* tem a necessidade de um interveniente extra que transmita as mensagens com as informações de cada um dos clientes, o que suporta, informações dos endereços e portas que podem ser contactados. Para encontrar os endereços e portas é usado o protocolo Interactive Connectivity Establishment (ICE) [35]. Este protocolo foi desenhado para funcionar mesmo através de *firewalls*, mas para alguns casos são necessários servidores externos que encaminham os pacotes (chamados Traversal Using Relay NAT (TURN) *servers*) [36].

O suporte desta API é grande por parte dos *browsers* Chrome, Firefox e Opera tanto para computador como para Android [37–39] . O Internet Explorer é o segundo *browser* mais usado do mercado [40], mas não tem planeado suporte [41].

Em Android o navegador web que é possível embeber dentro de uma aplicação atualmente não suporta WebRTC, só no futuro próximo (versão L da plataforma) estará disponível [42].

2.2.4 WebSockets

Outra forma de enviar dados de um *browser* é através de WebSockets [43]. Esta API é uma tentativa de trazer sockets semelhantes aos fornecidos pelo sistema operativo (Transmission Control Protocol (TCP) e User Datagram Protocol (UDP)) que operam segundo o modelo Open Systems Interconnection (OSI) [44] entre as camadas 4 (transporte) e 5 (aplicação). Os Websockets estão na camada aplicacional, têm todo um protocolo associado, funcionam sobre TCP e opcionalmente sobre Secure Sockets Layer (SSL) [43].

O protocolo usado pelo Websockets [43] foi concebido para ser compatível e funcionar sobre as mesmas portas de HTTP. Isto é conseguido através do *handshake* feito, em que é efetuado um pedido HTTP ao servidor a pedir a mudança de protocolo para Websockets.

As APIs existentes apenas permitem abrir uma ligação a um servidor destino (ou seja, funciona como um cliente), e não permitem criar um servidor local como nos *sockets* tradicionais fornecidos pelo o sistema operativo. Além disto não permite enviar dados por *multicast* (apenas sobre TCP). Estes fatores tornariam excessivamente limitativo o uso de Websockets no projeto para comunicação entre os clientes.

2.3 Protocolos para *multicast* fiável

Um dos pontos principais deste trabalho é o uso de *multicast* de forma fiável para partilha de informação. De seguida apresenta-se uma análise das soluções existentes para *multicast* fiável aplicáveis ao caso de uso descrito no capitulo 1.

2.3.1 Tree-Based Reliable Multicast (TRAM)

O TRAM [45] foi concebido para o envio de dados de uma fonte para múltiplos recetores localizados a mais de um salto de distância. Este protocolo forma uma árvore, em que no topo está o emissor e abaixo estão recetores. Alguns dos recetores ficam responsáveis por retransmitir pacotes. Estes recetores são chamados de *repair heads*. Os recetores normais ao receberem um pacote, enviam uma confirmação da receção por *unicast* para o seu *repair head*, este guarda em *cache* os pacotes até que todos os seus "filhos" confirmem a receção.

Os dados enviados têm um conjunto de dados adicionais como cabeçalho:

- Número de sequência: para identificar o pacote e permitir fazer a confirmação da receção deste;
- Identificador da sessão: para separar sessões que possam estar a usar o mesmo canal *multicast* e assim detetar colisões;

Existem ainda mensagens adicionais para informar o emissor do número de pacotes recebidos pelas sub-árvores, e *beacons* para informar do início e fim da sessão, e a sessão ainda continuar ativa, para caso o emissor não emita pacotes há algum tempo (caso contrário não seria possível distinguir entre o emissor estar sem transmitir ou ter saído).

O protocolo tem ainda ajuste da largura de banda da transmissão em casos de congestionamento. Mais detalhes sobre este podem ser encontrados em [45].

2.3.2 Light-weight Reliable Multicast Protocol (LRMP)

O LRMP [46] é um protocolo desenhado e pensado para entrega fiável de pacotes por *multicast* em ambientes que podem tolerar espera (não aconselhável para tempo-real). É desenhado para funcionar particularmente bem em grupos dispersos de recetores mas com possibilidade de haver subgrupos densos destes.

A técnica usada para recuperação de pacotes baseia-se no envio de Not Acknowledge (NACK). A deteção da perda de um pacote pode ser detetada pela falha nos números de sequência dos pacotes recebidos ou por o pacote de relatório enviado pela fonte. Após a deteção da perda é iniciado um temporizador para uma espera de tempo aleatório. Este mecanismo dá pelo nome de supressão de NACKs e serve para evitar colisões destes. Se receber um NACK para um pacote que não recebeu, o recetor atua como se ele próprio o tivesse enviado e vai para um período de espera. Caso contrário, verifica que o pacote ainda não foi recebido e se ainda for necessário envia o NACK. Por fim entra num período de espera em que reenvia o NACK após um dado tempo de espera, até obter resposta. Isto serve para colmatar o caso da fonte não ter recebido o NACK original.

O protocolo tem ainda um melhoramento em que tenta primeiro obter o pacote localmente (definindo um Time To Live (TTL) no cabeçalho do pacote IP baixo) e quando reenvia o NACK aumenta esse TTL. Mais detalhes podem ser encontrados em [46].

Assim sendo, qualquer recetor pode responder. Antes de responder o recetor faz um período

de espera também para evitar múltiplas respostas ao NACK. Para tal, cada recetor tem um espaço de armazenamento em que guarda os pacotes enviados e recebidos proporcional ao ritmo de envio dos pacotes.

O protocolo tenta ainda ter um bom comportamento na rede para evitar congestionamento. Para tal usa os pacotes recebidos dos recetores para se adaptar e ajustar o ritmo de envio dos pacotes.

Algumas características definidas são ainda a possibilidade de envio de pacotes sem fiabilidade (não têm garantias de entregas nem são detetadas perdas caso existam), relatórios seletivos dos recetores dos dados (são enviados com uma dada probabilidade que pode ser maior ou menor conforme o número de recetores). É ainda suportado o uso de pacotes Forward Error Correction (FEC), que consiste em adicionar pacotes redundantes que permitam reparar pacotes perdidos [47] sem necessidade de retransmissão. O uso destes pacotes é complicado porque podem adicionar tráfego extra desnecessário ou mesmo que necessário (por existirem perdas) não serem o suficiente para recuperar das perdas.

2.3.3 Pragmatic General Multicast (PGM)

O protocolo PGM [48] usa algumas técnicas vistas anteriormente, uso de NACKs para indicação de perda de pacotes que são enviados em *unicast* para a fonte de dados. Supressão de NACK através de confirmação da receção por parte de elementos de rede com suporte do protocolo que estejam entre a fonte de dados e o recetor, denominados de Network Elements (NE). Pacotes de correção FEC são também usados. Existe ainda a possibilidade de haver nós na rede designados para recuperar pacotes (denominados Designated Local Repairers (DLRs)).

Ao existir suporte por parte de nós na rede vários melhoramentos são efetuados, o mais notório é a confirmação da receção de NACKs denominados de NAK confirmation (NCF). Isto permite o que é denominado por antecipação do NACK, isto é, quando um NE já recebeu um NCF para um dado pacote, mesmo que mais tarde receba um NACK para esse pacote não vai enviar-lo para cima. Pois o NE superior já confirmou que recebeu um NACK para aquele pacote e vai encaminhar para a fonte. Os NE permitem ainda fornecer relatórios de congestionamento à fonte de forma a ajustar o ritmo de envio dos pacotes.

2.3.4 JGroups

JGroups não é um protocolo mas sim uma implementação de uma biblioteca para fazer algo semelhante ao que este projeto tenta. Foi analisado e testado, mas a implementação encontrada para Android [49] tinha vários problemas que causavam o incorreto funcionamento de *multicast*. Além disto o projeto é parco em documentação e detalhes do funcionamento do *multicast* fiável [50].

2.3.5 Conclusão

Dos vários protocolos analisados pode ser feito o curto resumo das principais características para obter fiabilidade e escalabilidade na tabela 2.1.

	TRAM	LRMP	PGM
Garante a entrega total da informação	Sim	Sim	Sim
Funcionamento em dados com necessidades de tempo real	Não	Não	Sim
Múltiplos emissores	Não	Sim	Não
Uso de NACK/Acknowledge (ACK) para deteção de perdas	ACK	NACK	NACK
Formação de grupos hierárquicos	Sim	Sim	Opcional
Suporte para pacotes FEC	Não	Opcional	Opcional

Tabela 2.1: Resumo das principais características dos protocolos para obter fiabilidade e escalabilidade

Atendendo a isto, apenas o PGM se enquadraria neste trabalho e mesmo este é bastante sobredimensionado em relação ao caso de uso. Tem vários elementos que não trariam vantagens no ambiente de uso, mas trariam desvantagens pelo custo das mensagens extra.

A maioria destes protocolos foram pensados para ambientes em que os recetores estão dispersos por várias subredes e contendo infraestruturas ou nós intermédios que permitem poupar largura de banda caso possa haver trocas de informação mais próximas. Dado que no nosso caso de uso toda a comunicação está assente em poucos ou até num só nó de rede que liga todos os intervenientes (emissores e recetores) estas soluções não têm qualquer ganho.

Para além disto, algumas das soluções não têm em conta a privacidade dos dados, mecanismos de autenticação ou gestão de novos recetores. Alguns deles não suportam também múltiplos

emissores e não têm isto em conta para a gestão da largura de banda.

Dadas as necessidades de privacidade dos dados, de permitir múltiplos emissores e de compatibilidade com vários sistemas operativos tomou-se como melhor opção desenvolver um protocolo mais simples, que usasse os conceitos partilhados pelos protocolos existentes estudados, mas que conseguisse suportar todos os requisitos iniciais.

Capítulo 3

Desenho da Arquitetura

Nesta secção vai ser discutido em primeiro a arquitetura que vai ser desenhada e cada um dos componentes da arquitetura. Quando necessário serão usados diagramas de sequência Unified Modeling Language (UML) para facilitar a analise dos protocolos e algoritmos [51].

3.1 Discussão da arquitetura

Das várias possibilidades analisadas na secção anterior foi escolhido apresentar às aplicações uma *framework* para o desenvolvimento sobre uma plataforma web. Ou seja, as aplicações podem correr num navegador da Internet, não estando limitadas a um sistema operativo/plataforma. Como tal faz sentido olhar para a possibilidade de uso de WebRTC.

3.1.1 WebRTC

Um das opções para ligar os utilizadores entre si foi o uso de WebRTC, como visto em 2.2.3 este permite comunicação de dados, vídeo e/ou áudio entre dois *browsers*.

Com o uso desta API ganhava-se uso de vídeo e áudio sem necessidades extras para lidar com a codificação de dados, pois os *browsers* já fazem isso. Dava ainda hipótese de qualquer aplicação já existente a usá-lo, poder ser adaptada para usar esta biblioteca.

Foi estudado a transmissão e os passos para estabelecer uma ligação, que se podem resumir nos seguintes:

1. Inicia o RTCPeerConnection com possibilidade de passar parâmetros opcionais (como por exemplo, o servidor para o protocolo ICE);

2. Adiciona *streams* de vídeo e/ou áudio ao RTCPeerConnection criado;

3. O primeiro cliente cria uma oferta da ligação no objeto RTCPeerConnection e envia para o outro cliente (mediante um servidor intermediário que sabe falar com ambos). Neste momento o *browser* começa a executar o protocolo ICE para descobrir endereços e portas e a cada par descoberto é invocado um *callback* a informar, estes dados devem ser também enviados para o outro cliente;

4. O segundo cliente recebe a oferta e cria uma resposta para ela. Quando pronta envia para o primeiro cliente pelo o servidor intermediário;

5. Os clientes trocam mensagens com os ICE Candidates (conjuntos de valores de endereço IP, porta e protocolo a usar (preferindo UDP em relação ao TCP)) até os clientes conseguirem ligar-se um ao outro;

As mensagens de oferta e resposta a esta seguem o standard Session Description Protocol (SDP) [52], exemplos podem ser encontrados em [53]. A destacar o facto de todas as sessões usarem obrigatoriamente DTLS-SRTP [54] e não haver controlo sobre qual os endereços e portas a usar, tornando assim impossível usar *multicast*.

Para tentar ultrapassar esse problema foi pensada a solução apresentada na figura 3.1. Cada cliente que participava na partilha de informação tinha a aplicação Web a correr no *browser* (Chrome, Firefox ou Opera para suportar WebRTC) e um serviço em Java. Este serviço funcionava como servidor Web (fornecendo a aplicação ao *browser*) e cliente WebRTC. O *browser* criava uma ligação com esta API para enviar dados (que podia estar a captar na câmara web, por exemplo). Por sua vez este serviço ao receber esta ligação tratava de enviar o fluxo de dados aos restantes clientes da sessão de partilha de dados.

Esta solução apesar de parecer exequível e criar a possibilidade de partilha de dados que um navegador Web conseguisse obter, tem vários problemas:

- **O WebRTC obriga a ter cifragem de extremo a extremo [55]**: é usado o protocolo Datagram Transport Layer Security (DTLS) [56] que usa Diffie-Hellman [57] para gerar as chaves para cifrar os dados. Isto significa que a chave nunca está em trânsito e é impossível obrigar a que uma chave seja a que queira-mos. Assim sendo entre o emissor e o seu serviço

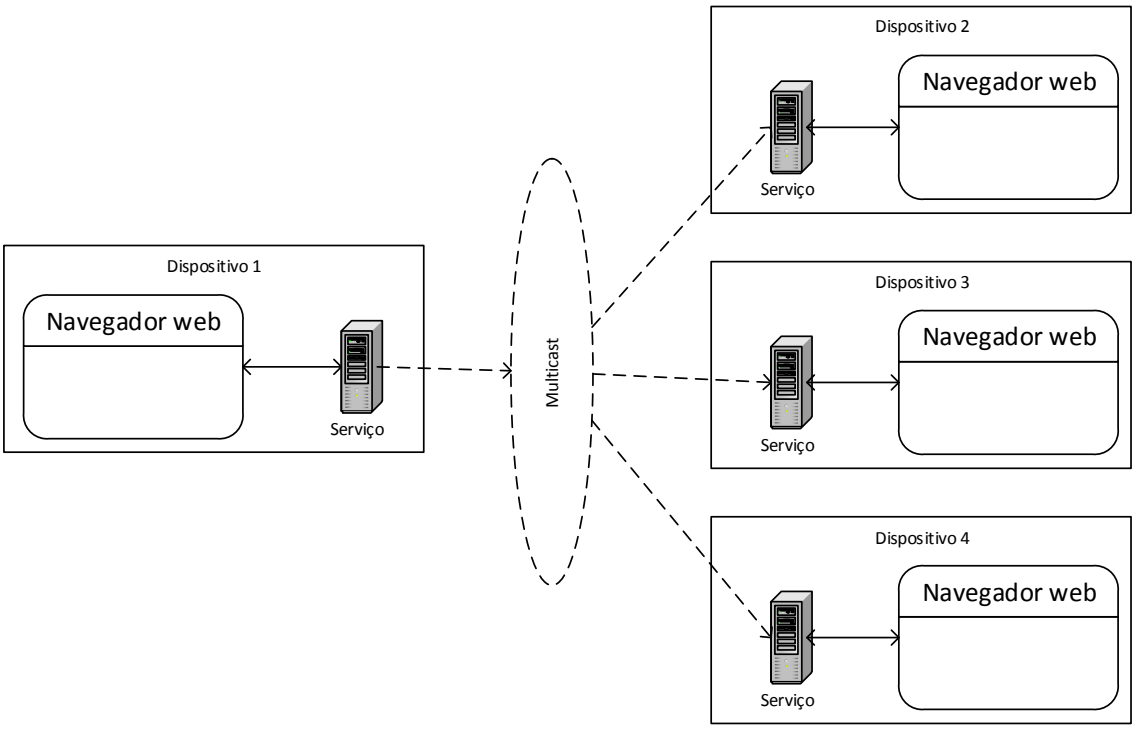


Figura 3.1: Arquitectura para uso do WebRTC

seria negociada uma ligação WebRTC que gerava uma chave, isto aconteceria para cada um dos clientes. O que no melhor caso obrigava aos dados a serem decifrados com a chave de WebRTC e cifrados com a chave do canal antes de serem enviados para os clientes por parte do emissor. Por sua vez cada um dos recetores ia ter de decifrar com a chave do canal de partilha de informação e cifrar com a chave da sua ligação WebRTC para fazer chegar os dados por este meio ao *browser*;

- **Vários protocolos necessários**: WebRTC usa obrigatoriamente SDP [58] para negociar a sessão [59], DTLS [56] para negociar a chave que vai ser usada para cifrar os dados e Real-time Transport Protocol (RTP) [60] para enviar os dados entre os clientes.

- **Limitação dos conteúdos disponíveis**: mesmo que os problemas anteriores não fossem só por si desencorajadores para usar esta tecnologia, o uso de WebRTC não resolvia o problema de fontes de dados além de câmaras web ou fontes de áudio disponíveis no *browser*. Fontes externas só acessíveis por USB ou Bluetooth iam continuar indisponíveis e teriam de ter soluções próprias.

3.1.2 Servidor Web e API HTTP

Apesar dos esforços para construir a aplicação inteiramente no navegador Web, atualmente os *browsers* não têm APIs para criar a aplicação necessária inteiramente sobre eles. Por isso, foi encontrada como solução a criação de um serviço ao qual o navegador Web pudesse chamar para enviar e receber dados por *multicast* de forma fiável (semelhante à figura 3.1).

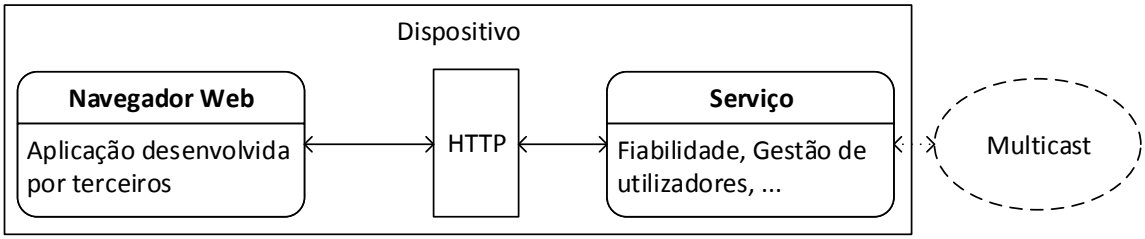


Figura 3.2: Aspeto do modelo idealizado

Desta forma foi idealizada uma uma estrutura modular representada na figura 3.2. Em que se teria o *browser* onde correm aplicações feitas para usar o trabalho desenvolvido para partilha de dados. Estas aplicações falavam com o serviço para partilha de dados por HTTP. Assim os componentes são:

- Serviço para partilha de informação via *multicast*: este serviço seria uma biblioteca responsável por a criação do ambiente de partilha de dados e envio destes de forma segura. É ainda responsável pela autenticação e gestão das chaves.

- Servidor Web: de pequenas dimensões e leve, de forma a poder ser usado em dispositivos Android sem grandes problemas de desempenho e dimensão [61]. Este permitia o serviço de partilha de informação e o *browser* comunicarem;

- HTTP Representational State Transfer (REST) API: uma API que permita através de um *broswer* utilizar de forma completa toda a biblioteca *multicast* construída que corre como serviço.

Esta estrutura terá de ser implementada em Java por razões de compatibilidade com Android como já discutido em 2.1, assim não só abre possibilidade de uso através de *browser* como também torna possível usar a biblioteca para partilha de informação em aplicações puramente Java.

3.2 Biblioteca *multicast*

Esta biblioteca tem como responsabilidade criar um ambiente para partilha de conteúdos, estes podem ou não ser em tempo-real de forma *multicast* com possibilidade de fiabilidade total. Tem ainda de conseguir gerir os utilizadores da sessão, isto é, tem de conseguir autenticá-los e autorizar estes a transmitir conteúdos ou não.

Para criar e gerir o ambiente de partilha, foi escolhido ter um coordenador centralizado denominado de gestor da sessão. Esta arquitetura ajusta-se ao nosso principal ambiente destinatário, podendo ser o professor que está a dar aula ou palestra, o gestor da sessão. Sistemas descentralizados também foram considerados, mas devido ao ambiente a que este sistema se destina a solução encontrada, aqui apresentada foi considerada superior. Não só se torna mais simples como não faz sentido existir uma sessão de partilha sem um gestor. Além disto, como o gestor é único na sessão, pode haver requisitos extra que podem ser controlados mais facilmente. Por exemplo, se o gestor da sessão precisar de mais poder de computação ou memória, isto pode ser tido em conta na escolha do dispositivo a ser usado.

3.2.1 Terminologia

Antes de continuar é importante definir alguns termos que serão usados:

- Sessão: é o ambiente sobre o qual um conjunto de utilizadores se ligam entre si e partilham informação;
- Gestor da sessão (GS): é o criador da sessão e responsável pela a autenticação, gestão de permissões na sessão, autorizar outros utilizadores a criar canais pelos quais podem transmitir e manter a sessão coerente entre todos os utilizadores;
- Canal: é uma abstração de um canal *multicast* por onde serão transmitidos dados para todos os utilizadores da sessão, em que pode ou não ter fiabilidade;
- Utilizador: este liga-se à sessão através da autenticação ao gestor da sessão e pode pedir para criar canais para transmitir dados;

3.2.2 Descoberta de sessões

Um dos componentes que fazem parte da biblioteca *multicast* é a procura e descoberta de sessões. Apesar de nos casos de uso definidos em 1.1 aparentemente configurar-se o cliente com um endereço de IP poder ser aceitável, torna-se pouco prático e complicado se considerarmos que podem chegar utilizadores que se queiram ligar à posteriori (um endereço IP pode mudar).

Um modo de anúncio e descoberta de sessões é portanto ideal. Dado ser utilizado *multicast* para transmissão de dados, este pode também ser usado para divulgar sessões existentes.

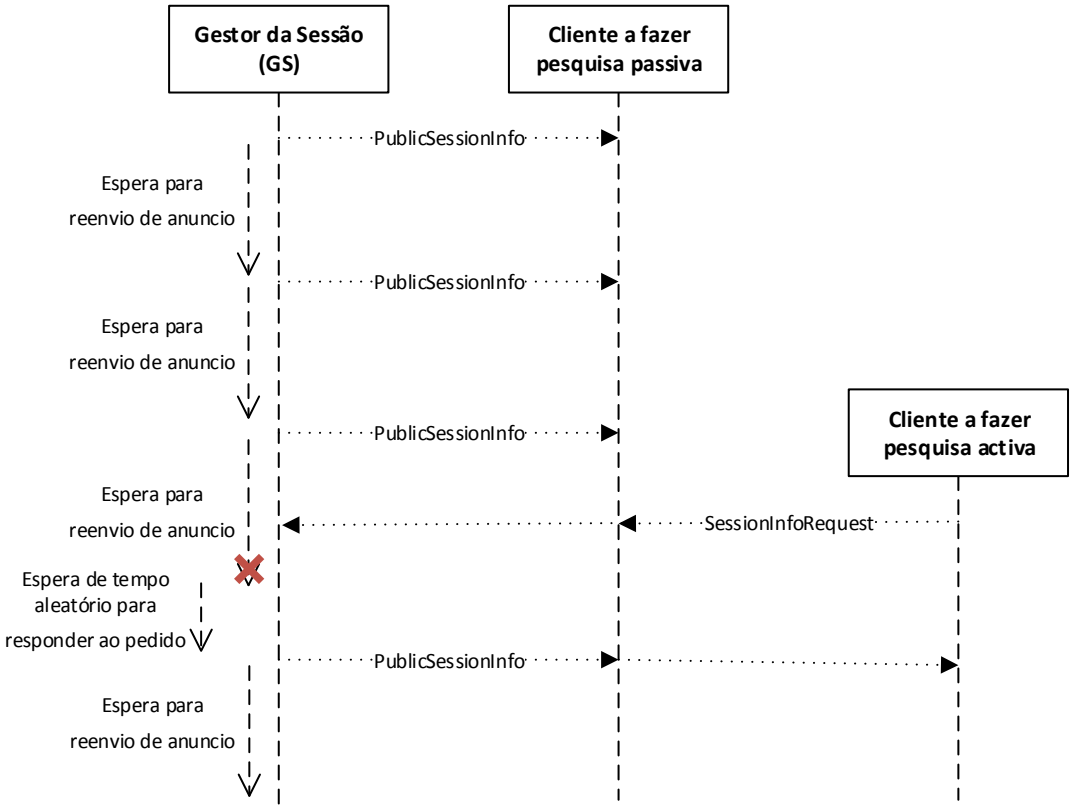


Figura 3.3: Diagrama de sequência de mensagens enviadas por cada um dos participantes na divulgação e descoberta de sessões em modo ativo e passivo. Todas as mensagens são enviadas por *multicast*

Assim sendo, foi desenhado o protocolo na figura 3.3, em que o Gestor da Sessão responsável pela manutenção da mesma anuncia periodicamente para um canal *multicast* a Sessão que mantém.

Quem quer descobrir sessões tem dois modos para o fazer. O modo passivo, em que escuta o canal em que as sessões estão a ser anunciadas, e o modo ativo, em que além de escutar como acontecia no modo passivo, envia uma mensagem para o canal de anúncio a pedir para

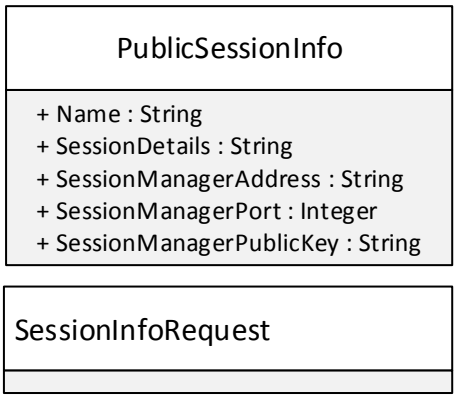


Figura 3.4: Objetos usados no protocolo de descoberta de sessões

anunciarem sessões. Os gestores das sessões ao receberem esta mensagem, esperam um tempo aleatório entre 0 e 10 segundos para enviar o anúncio da sua sessão. Esta espera serve para tentar evitar colisões de múltiplos anunciantes tentarem responder ao mesmo tempo.

A sessão anunciada tem a estrutura apresentada na figura 3.4, em que é anunciado o nome e detalhes definidos por quem cria a sessão, o endereço IP do gestor da sessão e a porta onde tem um servidor à escuta para autenticação, e a chave pública usada no processo de autenticação.

3.2.3 Autenticação

Dado ser um sistema centralizado o controlo da autenticação está no gestor da sessão. Este é responsável por todas as mensagens de controlo e por manter informações sobre todas as sessões de partilha de conteúdos, é portanto de elevada importância que quem se autentica a este possa garantir que está a falar com ele. Assim sendo, a autenticação do gestor da sessão é obrigatória. No entanto e dado nem sempre ser possível ter comunicação para o exterior, a verificação da identidade de quem se liga à sessão não é obrigatória. O protocolo realizado está descrito na figura 3.5.

Este protocolo consiste em:

1. O Cliente abre uma conexão ao Gestor da Sessão (GS);
2. O GS responde ao Cliente com a sua chave pública com a qual vai aceitar mensagens cifradas (esta deve ser igual à anunciada como descrito no anúncio da sessão na secção 3.2.2);
3. O Cliente responde com uma chave simétrica cifrada com a chave pública. Desta forma o

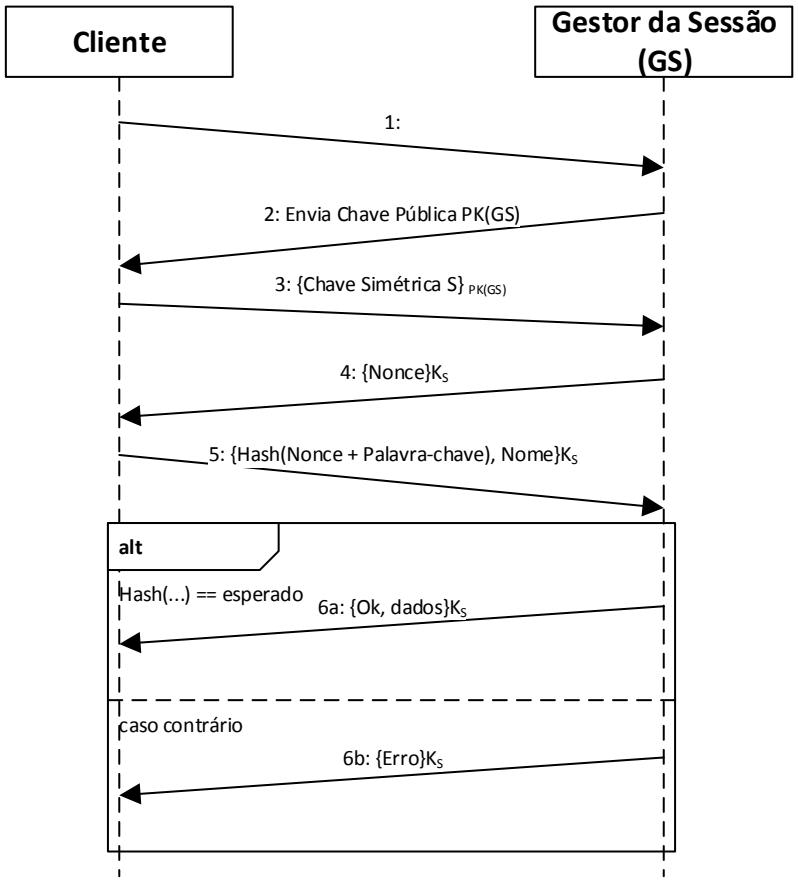


Figura 3.5: Diagrama de sequência das mensagens do protocolo de autenticação entre um Cliente e o Gestor da Sessão

Cliente fica comprometido com o servidor, pois a partir desta mensagem todas as seguintes serão cifradas com esta chave simétrica que apenas o GS e o Cliente, que se quer autenticar, conhecem;

4. O GS recebe a chave simétrica, decifra-a com a chave privada, verificando que foi corretamente usada a sua chave pública. A partir daqui todas as mensagens serão cifradas com a chave simétrica;

5. O GS envia um desafio ao Cliente (um número aleatório seguro);

6. O Cliente responde com o seu nome (que não deve ser tomado como seguro) e uma hash segura do resultado de concatenar o número aleatório de desafio com a palavra-chave de autenticação;

7. O GS verifica que a resposta está correta, em caso afirmativo responde com as informações da sessão, caso contrário um erro genérico. A informação enviada no caso de autenticação bem sucedida consiste em:

- Chave simétrica que está a ser usada durante a sessão para proteger as mensagens enviadas;
- Informações de todos os canais de envio de dados que existem em funcionamento;
- Informações do canal de controlo (bem como qual a última mensagem enviada por este), mais informações sobre este canal estão referidas em 3.2.5 .

3.2.3.1 Validação da Chave Pública

Um dos pontos importantes é validar a chave pública. O sistema só é seguro se os clientes conseguirem confiar na chave que o gestor da sessão envia no segundo passo. Na maioria dos sistemas é usado uma entidade que verifica a chave pública e assina a *fingerprint* desta. A chave ao ser divulgada leva a assinatura da entidade pública de confiança e os clientes dado conhecerem previamente a chave pública da entidade podem verificar a assinatura e confiar na chave [62, Capitulo 4.8].

Este método tem problemas na nossa infraestrutura, pois gerar uma sessão obrigaria a uma preparação prévia para obter uma chave pública e tratar do processo de assinatura desta. Tornando o sistema complexo e limitado, levando ao seu frequente uso sem esta medida de segurança.

Em vez disso optou-se colocar a *fingerprint* a ser verificada pelo utilizador. Uma *fingerprint* pode ser codificada de diversas formas, e tem tipicamente 128 bits ou 160 bits dependendo da função de *hashing* usada para a criar. Um exemplo de uma *fingerprint* de uma chave pública codificada em hexadecimal pode ser encontrado em 3.1.

$$\text{da39} : \text{a3ee} : \text{5e6b} : \text{4b0d} : \text{3255} : \text{bfef} : \text{9560} : \text{1890} : \text{afd8} : \text{0709} \tag{3.1}$$

Em vez de hexadecimal, por ser complexo para um utilizador comum transmitir e comparar todos os caracteres constituintes da *fingerprint*, foi escolhido codificar de outra forma mais fácil de comparar.

Obteve-se um dicionário ordenado por tamanho das palavras e alfabeticamente com 2^{16} palavras, de forma a codificar cada 16 bits numa palavra. Assim mapeia-se cada conjunto de 16 bits numa palavra, como exemplifica a figura 3.6. Com este meio de codificação tenta-se tornar mais fácil comparar *fingerprints*.

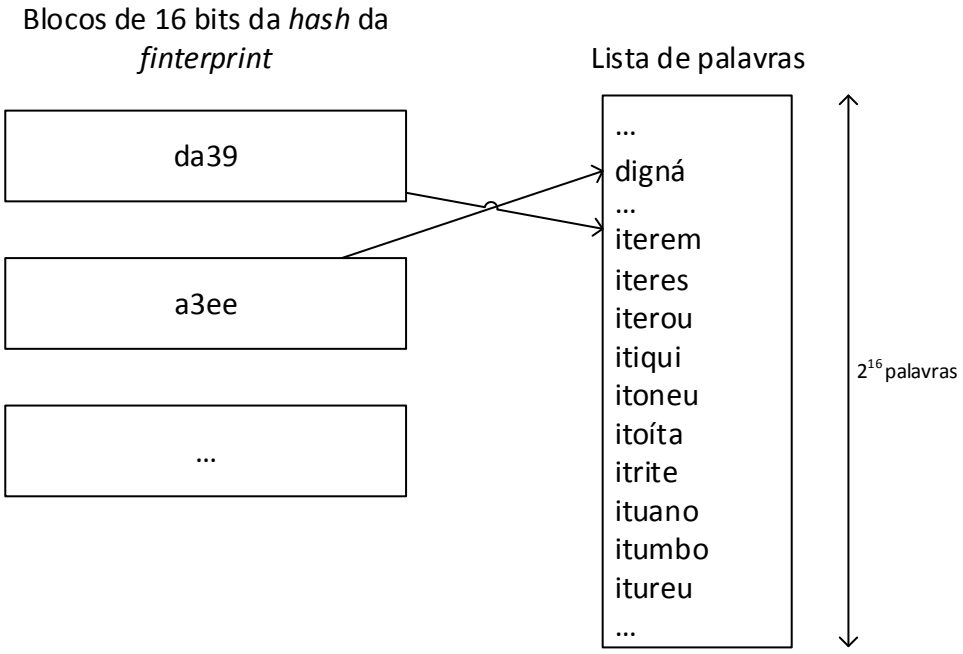


Figura 3.6: Conversão de *fingerprint* hexadecimal em palavras.

Atendendo que foi usada a função de *hashing* SHA-1 [63] por produzir um resultado de 160 bits, serão necessárias 10 palavras. Um exemplo do resultado da *hash* 3.1 codificada desta forma pode ser encontrado em 3.2.

iterem digná cherry papoce meio Iporá topaz delê cafulo tatuou (3.2)

Apesar de a escolha de SHA-1 pode comprometer a segurança do protocolo [64, 65]. Esta foi feita porque funções de *hashing* mais recentes têm *hashes* maiores, tornando o número de palavras maior.

3.2.3.2 Chave de autenticação do cliente

Aproveitando o uso de um dicionário para a verificação da chave pública, este foi também utilizado para obter a chave de autenticação do cliente à sessão. Esta chave é gerada por um gerador de números aleatórios seguro [66] num valor entre zero e 2^{16} mapeado no dicionário de palavras conforme foi feito na *fingerprint* da chave pública. Havendo assim um total de 65536 chaves possíveis.

Futuramente caso se ache que o tamanho da chave não é suficiente pode-se sempre acrescentar mais palavras (multiplicando assim o tamanho da chave e tornando mais difícil de atacar). Outra possibilidade é substituir completamente este método e deixar o gestor da sessão no momento da criação da sessão especificar a palavra-chave.

3.2.3.3 Gestão das chaves

Toda a informação enviada na sessão é cifrada por AES em modo ECB [67] com uma chave comum partilhada por todos os intervenientes. Esta chave é gerada por o Gestor da Sessão aquando da criação desta e distribuída aos clientes após a autenticação. O que os impede de abrirem livremente canais e transmitirem é o canal de controlo ser da posse do Gestor da Sessão. Todas as mensagens transmitidas no canal de controlo além de cifradas por esta chave que toda a gente conhece, são também assinadas com a chave privada correspondente da chave pública que foi usada no processo de autenticação e que todos os clientes conhecem. Na secção 3.2.5 podem ser encontradas mais informações sobre a criação de canais para envio de informação e do canal de controlo.

Apesar de noutros sistemas de partilha de conhecimento por *multicast* mudam as chaves quando da entrada ou saída de um elemento, nesta arquitetura consideramos que uma vez um utilizador esteja autenticado pode receber todos os dados daí em diante. Dado a autenticação passar por uma chave que já provou uma vez conhecer, pode posteriormente voltar a fazê-lo e obter assim a nova chave (caso a chave mudasse na eventual saída do utilizador). Caso haja mesmo a necessidade de excluir um utilizador de aceder aos dados após a sua saída, pode optar por criar uma nova sessão, com uma nova chave (da qual o utilizador não terá conhecimento) e os utilizadores da sessão anterior ligarem-se à nova sessão.

Considera-se também que um utilizador pode receber dados enviados previamente à sua entrada, assim desde que estivesse a capturá-los, mesmo sem chave para os decifrar, pode sempre fazê-lo a posteriori. Caso se queira evitar isto a solução passa por a criar uma nova sessão.

3.2.4 Fiabilidade em *multicast*

Um dos aspetos mais importantes de toda a biblioteca é a partilha de dados por *multicast* de forma fiável. Esta fiabilidade prende-se não só com a entrega de todos os dados, mas também com a manutenção da sua ordem.

Existem vários protocolos que conseguem isto conforme visto na secção 2.3, mas devido aos requisitos de execução em múltiplos dispositivos (em particular dispositivos móveis com elevadas limitações tanto em termos de memória como de processamento), seria difícil de encontrar implementações que funcionassem em todos os casos, ou com licenças que permitissem o livre uso e distribuição resolveu-se adaptar os conceitos num novo protocolo. Alguns protocolos foram ainda excluídos devido à enorme complexidade com casos de uso bastante diferentes do definido para este projeto.

3.2.4.1 Fiabilidade Total

De todos os protocolos estudados foram tiradas ideias que moldaram uma solução para alcançar fiabilidade total em *multicast*, algumas escolhas foram feitas tendo em conta soluções criadas em outros protocolos.

A fiabilidade na comunicação é conseguida por um de dois métodos, ou pela confirmação da receção (conhecido por ACK), usado por exemplo no TCP, ou através do envio de uma confirmação negativa (também denominado de NACK). A maioria dos protocolos que obtêm fiabilidade em *multicast* usam este segundo método, pois desta forma conseguem obter informação dos recetores sem sobrecarregar o meio com envio de ACKs para cada mensagem, ou grupos de mensagens se houver supressão como acontece no TCP [68].

O problema torna-se agora o envio dos NACKs. Uma opção seria enviar estes sobre um canal *multicast*. Isto traria o problema de sobrecarregar a rede com envio de pacotes *multicast* sem necessidade, pois o único destinatário importante é o transmissor do canal, que tem a obrigação de responder ao NACK. Por outro lado abria a possibilidade de outros recetores ao verem o NACK não pedirem também o mesmo pacote.

Outra solução é o envio do NACK por *unicast* (UDP) sem fiabilidade para o transmissor do canal. Desta forma só ele recebe, e a rede não é mais sobrecarregada no envio de mensagens além do necessário. Mas tem como problema os outros recetores que também tenham perdido o pacote, enviarem também um NACK. A perda do pacote pode ter acontecido no primeiro salto, o que causaria que nenhum recetor recebesse o pacote. Isto causaria colisões e podia nenhum NACK chegar ao destino.

A solução encontrada passa por enviar os NACKs por UDP diretamente para o transmissor do canal, mas antes de os enviar esperar um tempo aleatório num intervalo pré-definido. Isto

permite não só evitar colisões (pois cada recetor que perdeu o pacote vai esperar um tempo diferente), como também pode acontecer o caso de entretanto o pacote perdido chegar, pois pode ter simplesmente sofrido um atraso ou já alguém ter pedido e o emissor o ter reenviado.

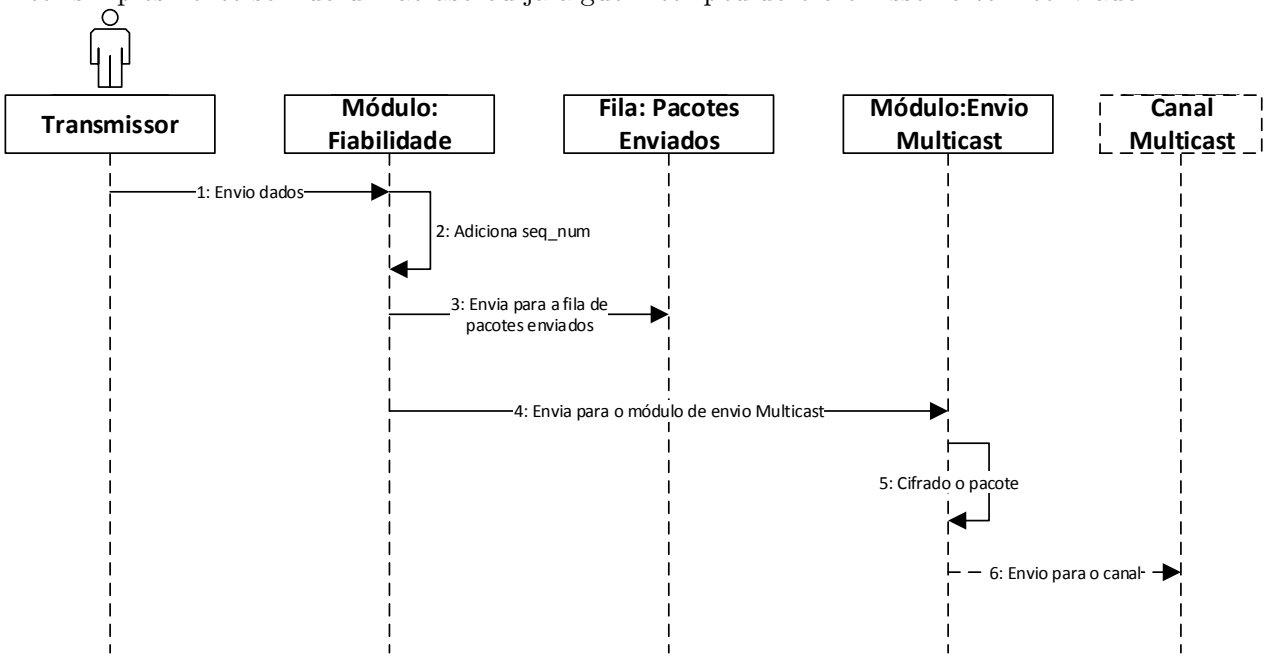


Figura 3.7: Diagrama de sequência do protocolo de fiabilidade do ponto de vista do emissor

Assim sendo o protocolo resultante do ponto de vista de quem envia está representado na figura 3.7, e pode ser descrito da seguinte forma:

1. O dono do canal envia para o módulo de fiabilidade os dados que quer enviar;
2. É adicionado um número de sequência para se conseguir manter a ordem e detetar a perda de pacotes;
3. O pacote com os dados e o número de sequência é copiado para uma fila que guarda todos os pacotes enviados, podendo assim reenviar o pacote caso seja dado como perdido por algum recetor;
4. O pacote é enviado para o módulo de envio para o canal *multicast*;
5. O pacote é cifrado e enviado para o canal *multicast*;

Para quem recebe, o processo está representado no diagrama da figura 3.8 e pode ser descrito como:

1. Um módulo está constantemente a receber do canal *multicast*, ao receber uma mensagem decifra-a e envia para o módulo de fiabilidade;

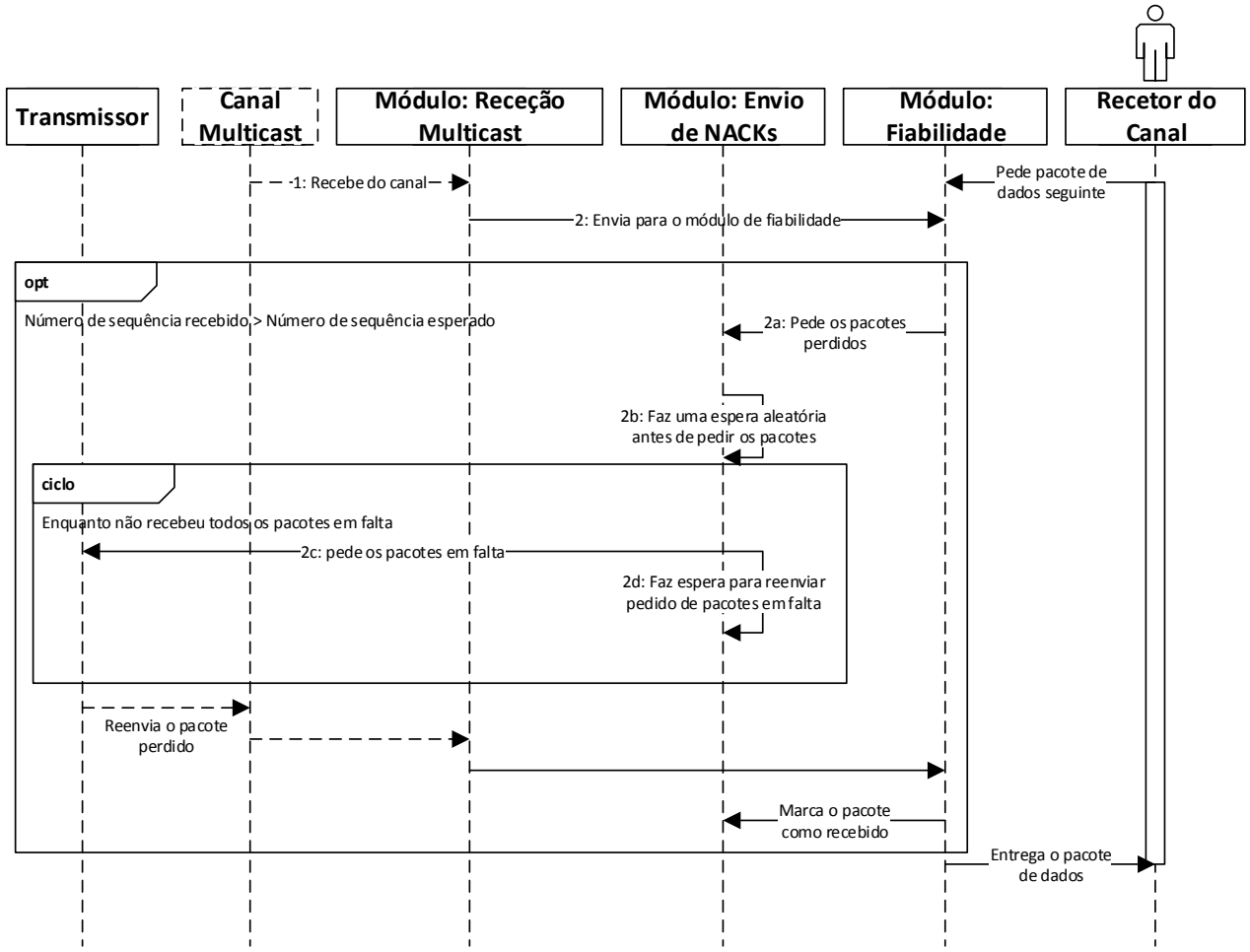


Figura 3.8: Diagrama de sequência do protocolo de fiabilidade do ponto de vista do recetor

2. No módulo de fiabilidade começa por verificar se o número de sequência é igual ao esperado. Se for inferior descarta o pacote, pois trata-se de uma repetição de outro já recebido. Em caso afirmativo envia o pacote para a camada superior, juntamente com todos os outros que já recebeu que têm número de sequência superior, atualizando o número de sequência seguinte esperado. Caso contrário recupera os pacotes perdidos da seguinte forma:

   (a) Calcula quais os pacotes perdidos, estes são os pacotes desde o número de sequência do pacote seguinte esperado até ao número de sequência do pacote recebido;

   (b) Inicia um relógio de tempo aleatório entre 0 e 10 segundos de espera antes de pedir os pacotes;

   (c) No final desse tempo de espera, caso não tenha ainda recebido todos os pacotes perdidos envia um pacote NACK por UDP para o transmissor do canal. O NACK contem o número de sequência de todos os pacotes em falta, evitando o envio de um NACK por pacote perdido;

(d) Inicia um novo relógio com um tempo aleatório entre 30 e 40 segundos e espera esse tempo;

(e) Volta a enviar o NACK de todos os pacotes em falta;

(f) Volta ao ponto 2d até todos os pacotes em falta serem recebidos;

3. O recetor do canal que estava em espera por o pacote seguinte de dados recebe resposta;

3.2.4.2 Fiabilidade moderada para uso em tempo-real

No caso de uso para transmissão de dados em tempo real a solução apresentada não tem em conta os atrasos que podem existir entre o envio de pacotes e a receção quando há perdas de algum pacote. Isto causa uma má experiência, podendo até quebrar completamente a sensação de transmissão de dados em tempo-real. O tempo até recuperar pacotes perdidos pode ser grande, e mesmo que a perda tenha sido a dados já relativamente distantes temporalmente, não há avanços enquanto estes não chegam.

Foi portanto adaptada uma solução em relação há fiabilidade total do ponto de vista do recetor, como demonstra o diagrama da figura 3.9. O envio de pacotes não sofre alterações em relação ao anterior, no entanto os recetores têm um cuidado acrescido ao tentar recuperar pacotes. Caso passe mais de um dado valor temporal após a deteção da perda, o pacote é esquecido e passado à frente. Este valor pode ser ajustado aquando da criação do canal.

Cuidados especiais sobre os dados enviados devem ser tidos em conta, pois estes podem não conseguir ser lidos quando há falhas. Este cuidado não é abordado neste trabalho, é assumido que quem utiliza esta biblioteca para o envio de dados escolhe os parâmetros de acordo com o tipo de dados.

De notar que a escolha da fiabilidade fica a cargo da aplicação que cria o canal, esta deve saber escolher entre ter fiabilidade total ou moderada para tempo-real tendo em conta os dados e a situação de uso.

3.2.5 Manutenção da sessão

Um dos pontos importantes para partilhar informação em grupo é manter uma visão igual de toda a sessão entre todos os clientes. No caso presente os únicos elementos que se alteram são os canais existentes, podendo ser fechados canais existentes ou abertos novos consoante as necessidades.

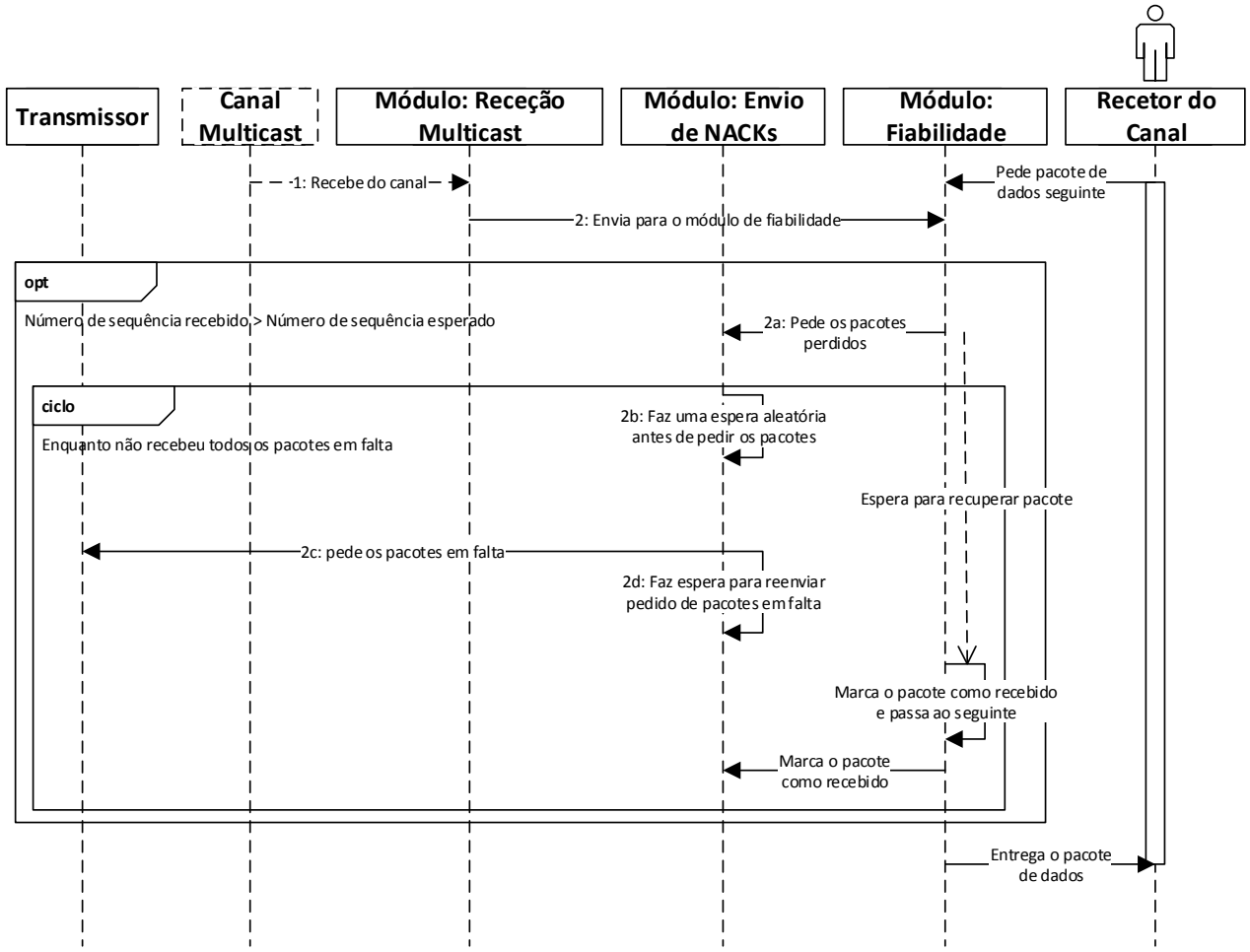


Figura 3.9: Diagrama de sequência do protocolo de fiabilidade para tempo-real do ponto de vista do recetor

Opcionalmente podiam também ser partilhadas informações da entrada de novos utilizadores. A saída de utilizadores não seria possível devido à complexidade de detetar a mesma.

A dificuldade de deteção da desconexão de um participante deve-se ao fato de não haver conexões a tempo inteiro com todos os utilizadores da sessão, não é possível saber quando algum se ausenta. Uma possibilidade seria ele informar da sua saída, mas aí estaríamos a descuidar casos em que ele ficou sem possibilidade de comunicar essa informação. Por exemplo, quando um terminal fica sem energia elétrica.

É necessário comunicar pelo menos as informações para todos terem oportunidade de receber todos os dados transmitidos de todos os canais de informação. Para isto ser possível foi desenhado um protocolo em que todos os transmissores antes de criarem um novo canal pedem ao Gestor da Sessão autorização para isso. Isto serve dois propósitos, por um lado o Gestor da Sessão tem todo o controlo sobre quem pode transmitir e pode autorizar ou não a criação do canal.

Por outro, existe um ponto central que pode coordenar as informações dos canais existentes e informar os utilizadores sobre esses canais.

Para informar os utilizadores da sessão destas "novidades" foi criado um canal de controlo.

3.2.5.1 Canal de Controlo da Sessão

O canal de controlo criado tem um funcionamento semelhante ao canal de fiabilidade descrito na secção 3.2.4. Todos os utilizadores da sessão começam à sua escuta após se liguem à sessão e é nele que são transmitidas mensagens a informar da criação ou eliminação de canais de informação.

Para evitar que qualquer um que saiba a chave simétrica, com a qual todos os pacotes são cifrados, transmita uma mensagem para abrir um canal, todos os pacotes neste canal são assinados pela chave pública do Gestor da Sessão. Ou seja:

$$\{[\text{Mensagem}]_{\text{PK(GS)}}\}_{\text{S}} \tag{3.3}$$

Sendo $[Mensagem]_{PK(GS)}$ a ação de assinar com a chave privada correspondente da chave pública do gestor da sessão e S a chave simétrica partilhada por todos os utilizadores da sessão.

Dado serem enviadas poucas mensagens neste canal, apenas quando existe a criação ou remoção de um canal, foi acrescentada um reenvio automático da última mensagem de 15 em 15 segundos[1]. Desta forma é mais fácil haver deteção de mensagens perdidas, e caso só a última mensagem esteja em falta esta é de imediato recuperada. Para os clientes que já a tinham recebido, nada muda, uma vez que o número de sequência da mensagem seguinte esperada é superior ao da mensagem recebida, o que leva a que a mensagem seja descartada, conforme descrito na secção 3.2.4.

3.3 Servidor Web

Para tornar a biblioteca desenvolvida disponível a aplicações Web que podem correr num navegador de Internet é necessário o uso de um servidor web. Este servidor torna possível o *broswer* interagir com código que corre na máquina virtual Java. Tornando assim possível o acesso aos conteúdos recebidos através de uma API baseada em HTTP.

[1]Este e outros tempos utilizados no protocolo de fiabilidade foram baseados nos protocolos analisados na secção 2.3.

O servidor em questão foi pensado para ser o mais simples possível e leve, tendo apenas o essencial para ser capaz de suportar toda a API HTTP que foi desenhada. Como tal, o servidor deve ter *bindings* para cada caminho do Uniform Resource Locator (URL) para uma classe Java poder tratar do pedido. Fazendo desta forma a delegação dos pedido de um caminho para a API criada, mas os restantes serem tratados como pedidos a ficheiros numa pasta. Desta forma facilita-se a adição de páginas e ficheiros adicionais (*scripts* em Javascript ou folhas de estilo CSS, por exemplo) para ser fácil criar toda uma aplicação Web e distribuir com o servidor e a biblioteca desenvolvida. O servidor será apresentado com maior detalhe na secção 4.2.

3.4 API HTTP

Para expor a biblioteca criada para envio de informação por *multicast* com fiabilidade ao navegador de Internet é preciso existir uma API sobre HTTP. Esta é a forma mais simples de o fazer, pois HTTP é o protocolo mais falado por esta aplicação, existindo APIs em Javascript para usar todas as suas capacidades para comunicar com um servidor. APIs essas que são standard e amplamente suportadas em todos os *browsers* [69].

A API foi desenhada com o intuito de disponibilizar todo o potencial da biblioteca *multicast* previamente definida.

3.4.1 Descoberta de sessões

Um dos elementos iniciais a disponibilizar deve ser a descoberta de sessões, pois este é um dos pontos de entrada para descobrir e ligar-se a uma sessão existente. A API para o fazer é a apresentada na figura 3.10. Sendo possível iniciar a pesquisa com um pedido POST de modo a obter as sessões conhecidas e mais informações para cada uma delas.

Dado esta API permitir ligar-se a uma sessão, foi criado o atalho para além de poder referenciar a sessão pelo nome, poder usar a palavra "actual" como atalho. Desta forma a aplicação não tem de guardar esta informação e para facilitar manter estado pode perguntar à API pelos dados da sessão atual.

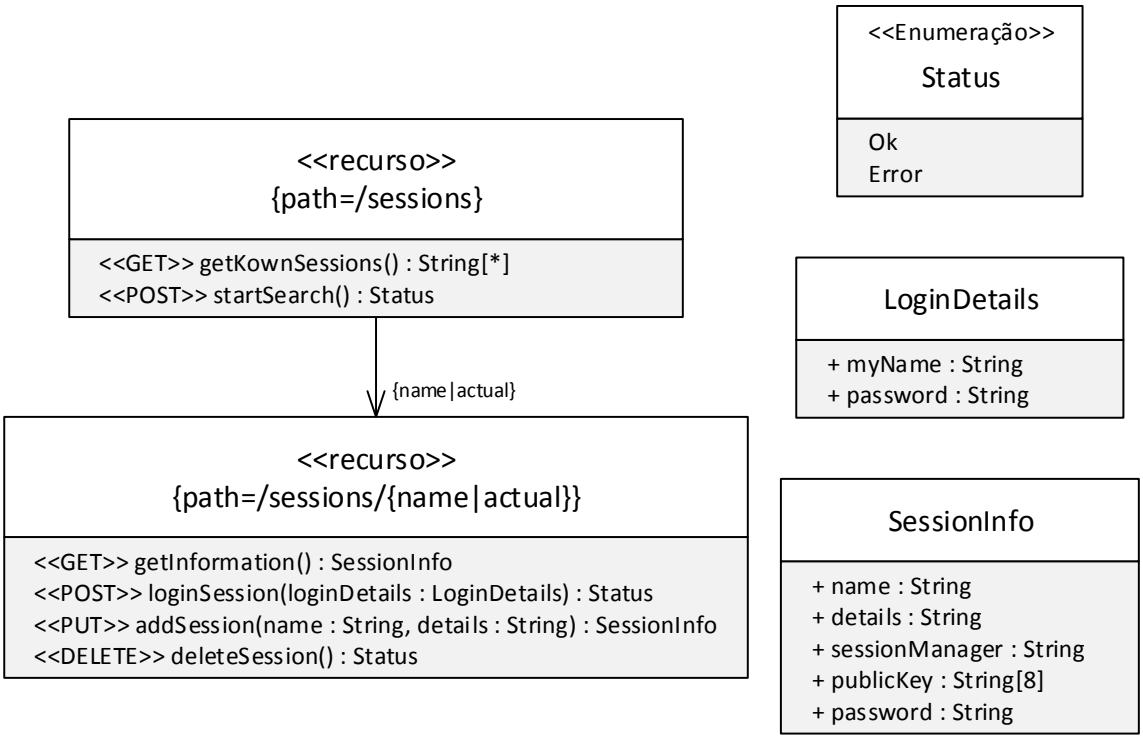


Figura 3.10: API via HTTP para descobrir e autenticar em sessões

No diagrama o *path* indica o URL do recurso. Cada método tem a indicação do tipo de pedido HTTP que deve ser invocado no URL para executar a função, os parâmetros que aceita e qual o resultado do pedido. É definido entre dois recursos qual sufixo no URL que deve ser acrescentado para passar para o outro recurso.

3.4.2 Criação de sessão

Para haver sessões a descobrir é necessário haver forma de alguém as criar. Assim e sobre a API apresentada na figura 3.10 é possível fazer um pedido HTTP PUT para adicionar uma sessão, sendo retornadas informações da sessão, em particular a palavra de autenticação para ser distribuída por o gestor da sessão aos clientes que se devem ligar.

3.4.3 Canais para envio de informação

Após a criação ou conexão a uma sessão é necessário operar sobre os canais de dados existentes. Os canais podem ter ou não fiabilidade, mas isso é transparente na leitura ou escrita dos dados para eles.

Para manipular os canais foi criada a API da figura 3.11. Usando esta API é possível saber

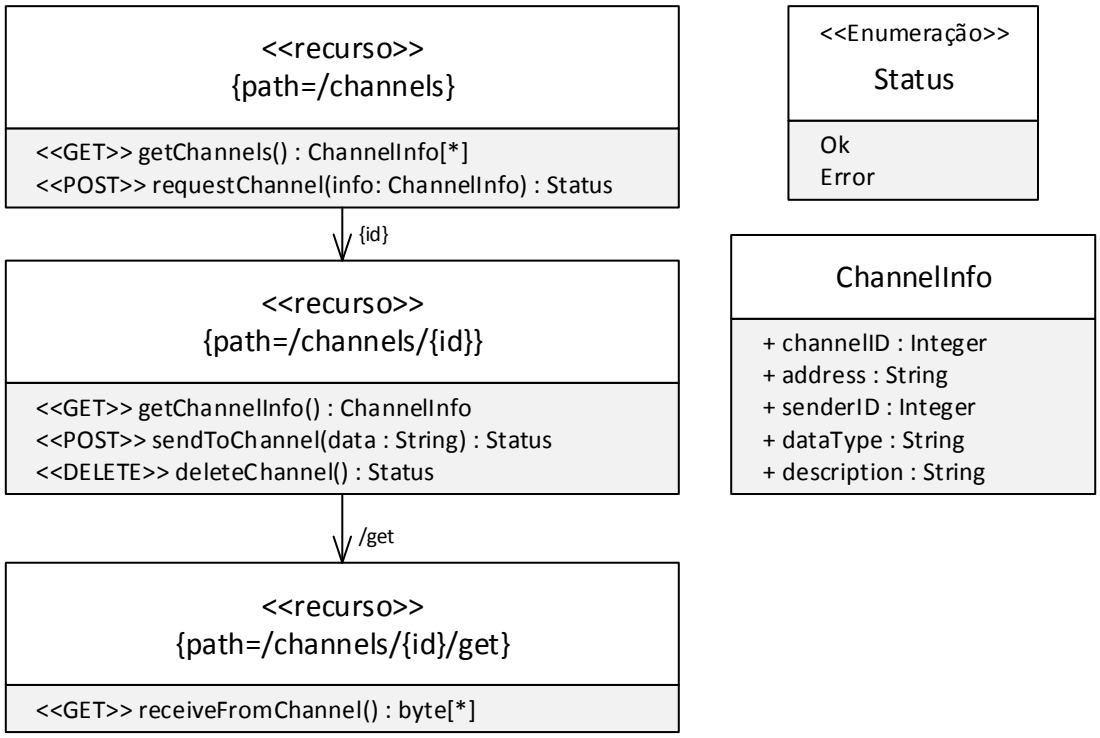


Figura 3.11: API via HTTP para criar, descobrir, enviar e receber dos canais

quais os canais existentes, informações sobre os canais, receber ou enviar dados se for cliente ou transmissor. Permite ainda eliminar o canal, eliminando-o apenas localmente se for cliente ou extinguindo-o completamente se for o transmissor.

3.4.4 Notificações

Para evitar ter de estar repetidamente a obter a lista de canais para detetar a adição de novos, faz sentido existir uma forma do servidor notificar a aplicação no *browser* dos acontecimentos recebidos no canal de controlo. Por esta razão foi criada uma API de notificações. Esta API permite o envio da mensagem *Notification* ilustrada 3.12 quando ocorre algum evento significativo.

O seu modo de funcionamento está descrito com mais detalhe na figura 3.13.

Dado algumas notificações obrigarem a uma interação da aplicação com o servidor, por exemplo para autorizar a criação de um canal, foi também criada uma API para a aplicação responder. Para isso é adicionado um identificador a cada notificação, um número único, incrementado a cada notificação e definida na API da figura 3.14. É também adicionado no servidor Web a possibilidade de adicionar uma função a ser chamada na biblioteca, quando esta envia a notificação do evento para o servidor Web.

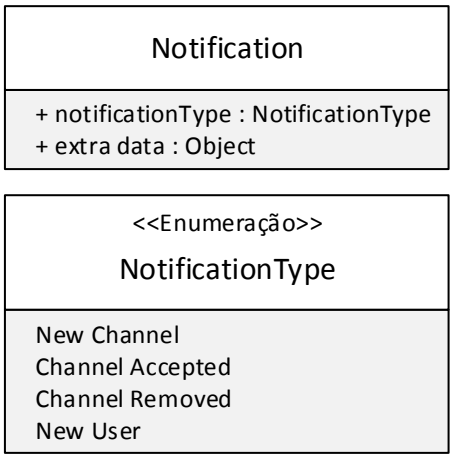


Figura 3.12: Tipo de mensagem das API de Notificações

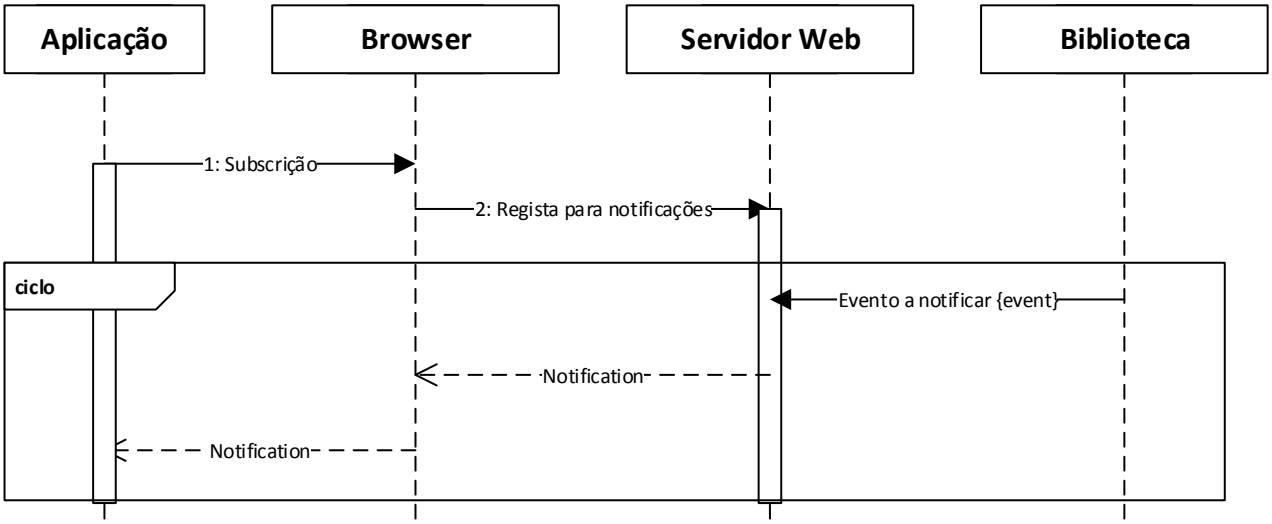


Figura 3.13: Diagrama de sequência do funcionamento das Notificações

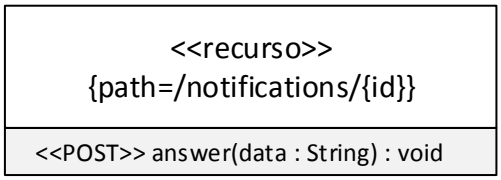


Figura 3.14: API via HTTP para responder a notificações

3.4.5 *Plugins*

Para estender as capacidades possíveis do Web *browser* foi ainda desenhado um sistema de *plugins*. Estes *plugins* são componentes que correm na máquina virtual (VM) de Java ao lado do servidor Web e que podem ser úteis para obter, processar ou armazenar dados enviados ou a enviar.

Desta forma abrem-se as possibilidades de recolha de dados a novos dispositivos. Por enquanto

um web *browser* não tem ainda capacidades para se ligar a dispositivos por USB ou Bluetooth de forma genérica. Do lado da VM de Java é possível ligar a estes dispositivos e recolher dados, tornando necessário este componente.

Pelo facto de os dados estarem a ser recolhidos na VM e o canal de envio estar também na mesma, faz sentido adicionar ainda a possibilidade de tratar os dados sem estes terem de passarem no *browser*. Isto permite não só facilitar o desenvolvimento para novas fontes de dados, como também diminuir a latência que podia advir entre enviar os dados para o *browser* e voltarem à VM para serem enviados para o canal.

Para além de entrada e processamento de dados, definiu-se ainda outro tipo de *plugin*, o de saída. Este pode servir para armazenar dados ou até enviá-los para um servidor remoto para serem processados.

O funcionamento destes *plugins* deve permitir ter múltiplos leitores para uma fonte. Pois só assim é possível ter um a recolher, por exemplo, de um dispositivo Bluetooth e estar conectado a ele um que envie para um servidor para armazenamento e arquivo dos dados, e outro a enviar para um canal *multicast* fiável.

Já do ponto de vista de entradas, cada *plugin* deve poder especificar o número máximo de entradas que permite. Quando esse número não é especificado, existe apenas uma.

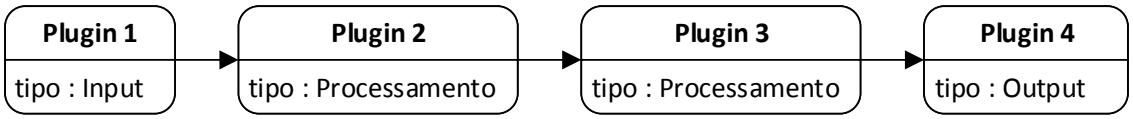


Figura 3.15: Exemplo de um grafo de *plugins* simples

Do ponto de vista de funcionamento o conjunto dos *plugins* forma um grafo como o representado na figura 3.15, cujo funcionamento é descrito no diagrama da figura 3.17. Neste exemplo, o funcionamento é bastante simples pois cada *plugin* envia simplesmente para o seguinte, mas no exemplo da figura 3.16 o *Plugin 2* e o *Plugin 3* ambos recebem o que o *Plugin* 1 produz, o *Plugin 4* aceita dados do *Plugin 2* e *3* produzindo dados lidos pelo *Plugin 5*.

Cada *plugin* internamente tem uma função definida pelo implementador, e que é chamada a cada passo de ciclo enquanto existem dados. O *plugin* indica quando deixa de haver mais dados. Isto é necessário para saber quando deve parar-se de executar o *plugin*. Nos *plugins* que

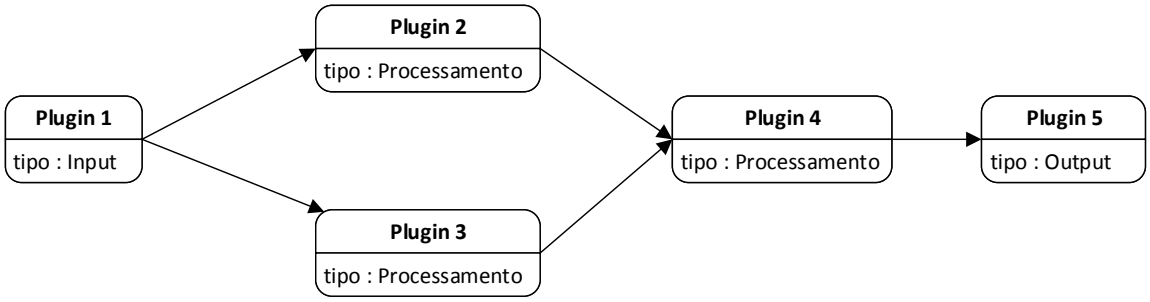


Figura 3.16: Exemplo mais complexo de um grafo de *plugins* em que existem dois que lêem de uma fonte e um *plugin* que aceita duas fontes de dados

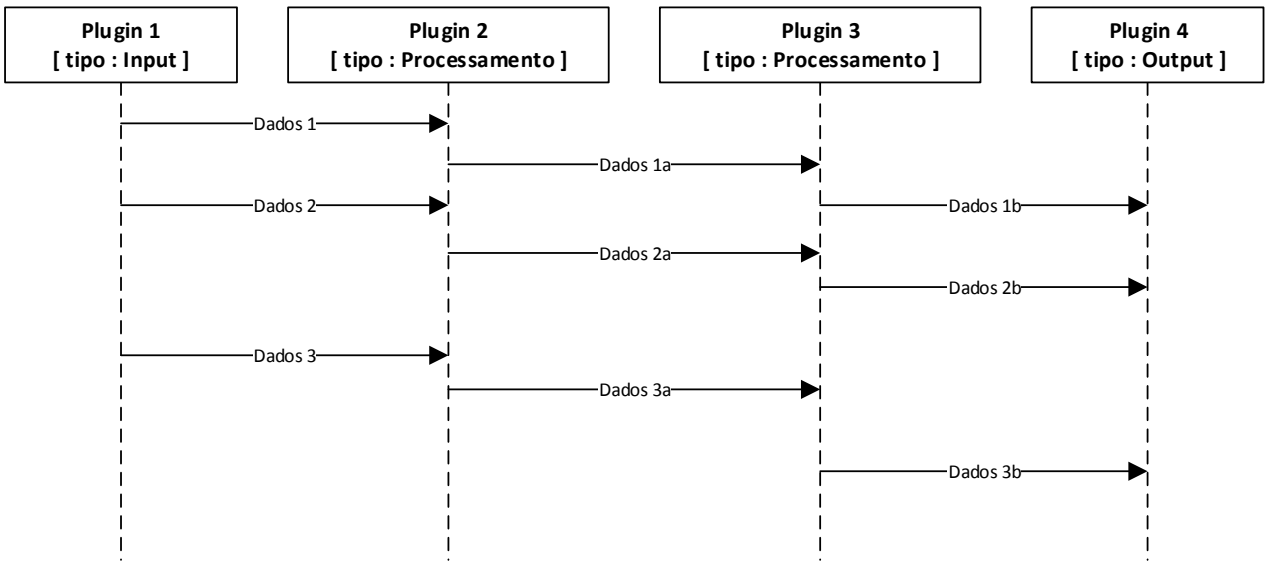


Figura 3.17: Diagrama de sequência do funcionamento de múltiplos *plugins*

processam os dados recebidos ou enviam para o canal *multicast* isto pode ser detetado facilmente, por deixar de haver mais dados a serem recebidos e fontes a produzi-los. No caso de *plugins* de origem de dados isto não é possível, razão pela qual foi adotada esta abordagem que salvaguarda os vários casos possíveis de uso.

O ciclo de vida de um *plugin* começa com um pedido numa API via HTTP para criar um grafo com o *plugin*. Dado os *plugins* terem sido pensados para conseguir o que não seria possível fazer num *browser*, ou simplesmente não seria o ideal a fazer, faz sentido as operações de criação, eliminação, edição de um grafo de *plugins* serem apresentados desta forma.

A API definida para criação e manipulação de grafos pode ser vista na figura 3.18.

Para no *browser* conseguir obter uma lista de *plugins* existentes (nomes deles) e informação

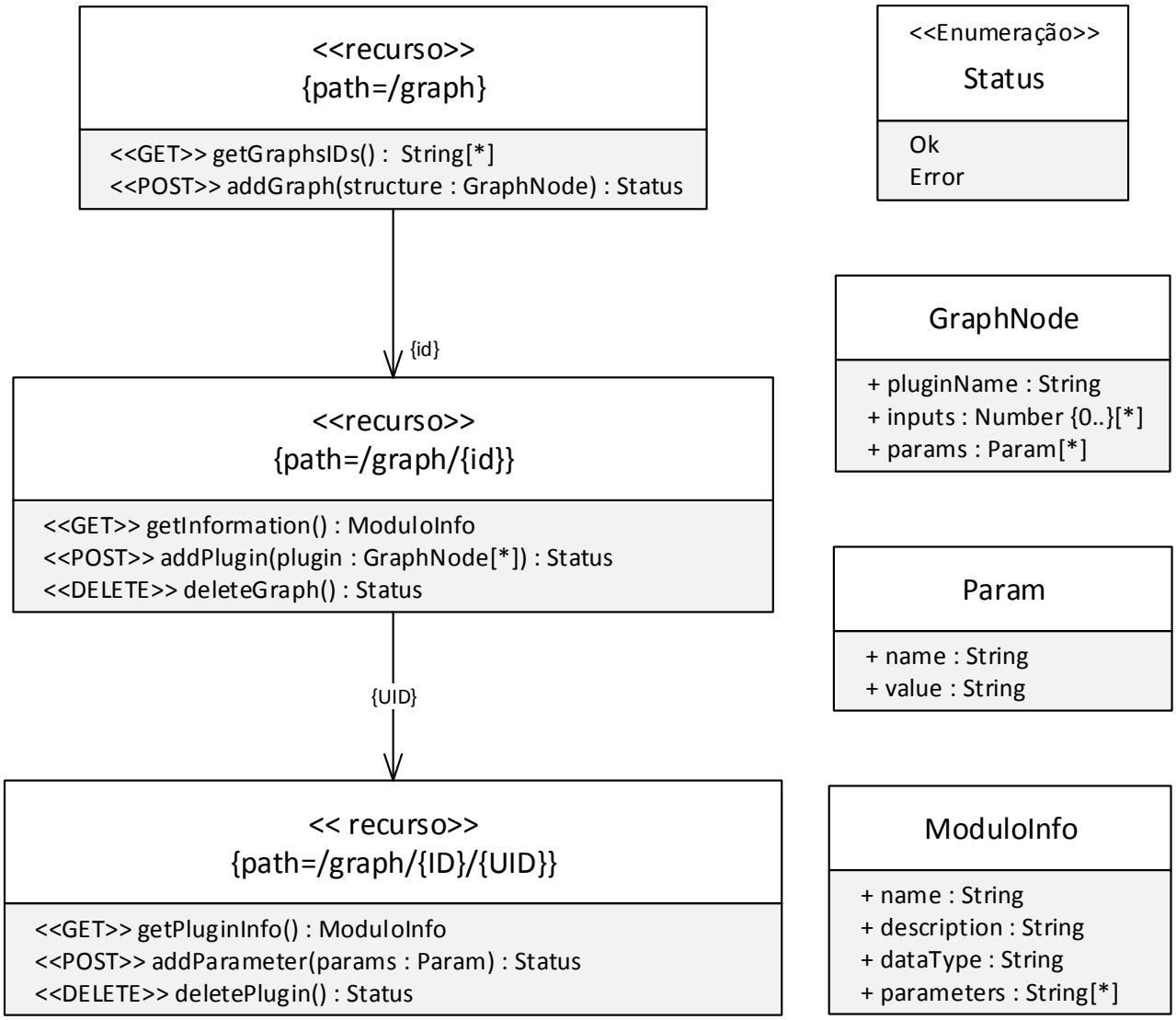


Figura 3.18: API via HTTP para criar e manipular grafos de *plugins*

adicional sobre eles (descrição, tipo de dados que suportam, parâmetros) foi também criada uma HTTP API descrita na figura 3.19.

3.4.6 Visualizadores

Além dos *plugins* foi criada também uma forma de tornar mais simples a reutilização de componentes, em particular visualizadores de conteúdos, por parte das aplicações.

Para isto foi pensada a ideia de visualizadores, que são componentes auto-contidos que permitem visualizar o conteúdo de um canal. Estes componentes correm inteiramente no *browser*, podendo usar HTML, CSS, Javascript e *plugins* de terceiros (por exemplo flash) para mostrar o conteúdo. Dado serem componentes que não têm de ser compilados (ao contrário dos *plugins*,

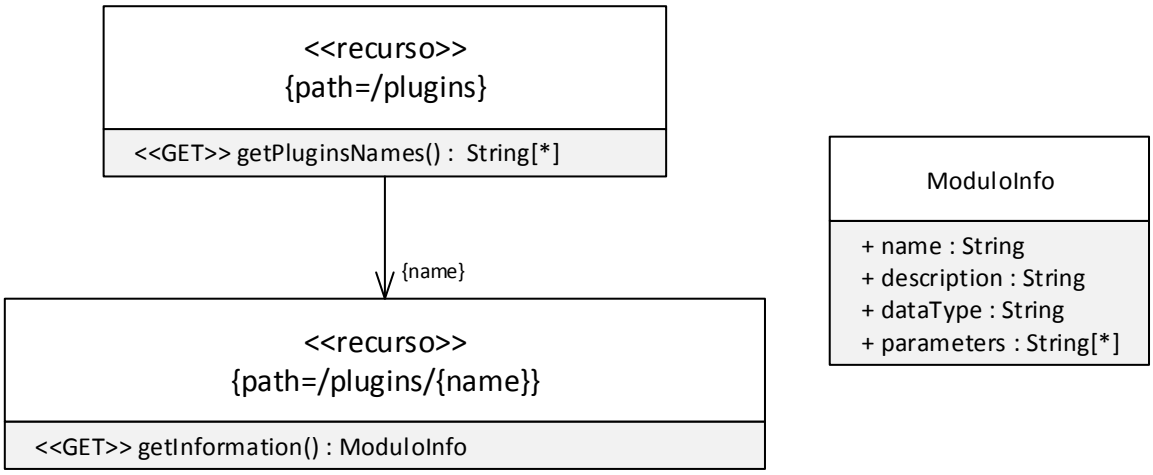


Figura 3.19: API via HTTP para obter informações dos *plugins* existentes

que correm na VM de Java), os visualizadores podem ser adicionados à posteriori. Foi criada uma API (ver figura 3.20) para facilitar a gestão de visualizadores: adicionar, listas existentes e carregar o visualizador.

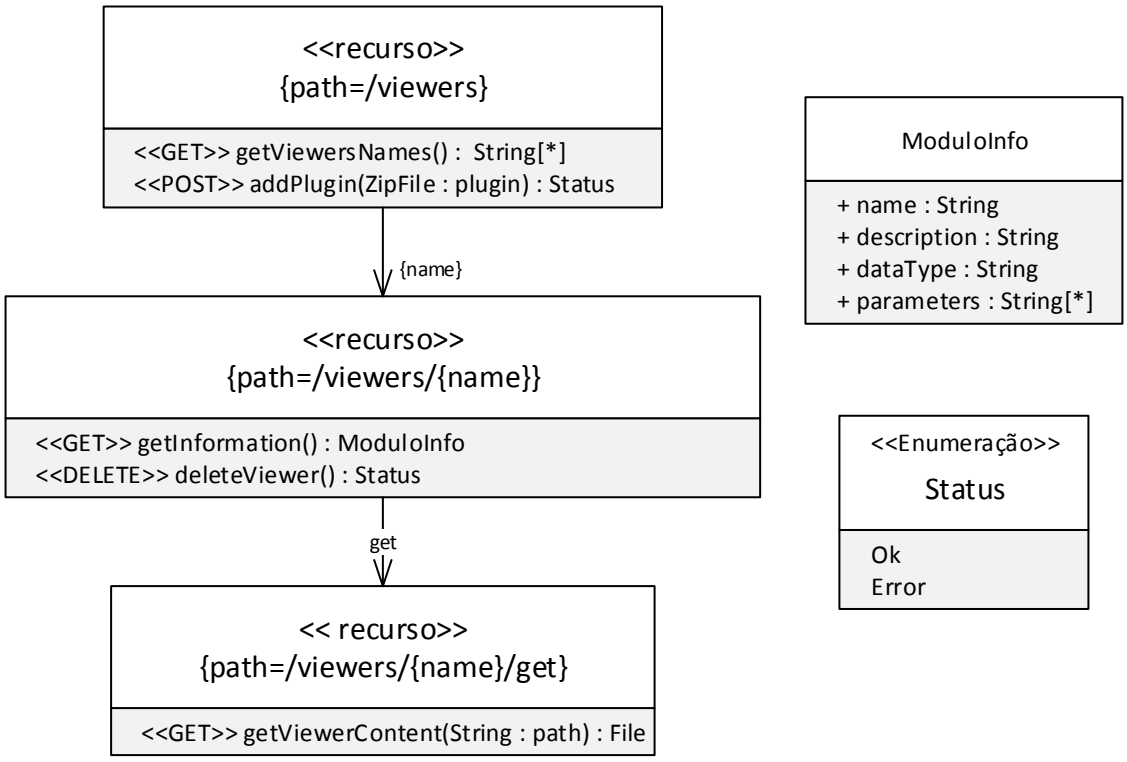


Figura 3.20: API via HTTP para adicionar, obter informações e carregar visualizadores existentes

Para tornar mais simples o envio e obtenção dos visualizadores optou-se por definir que estes devem estar armazenados num ficheiro ZIP. Desta forma todo o seu conteúdo está contido num

ficheiro, evitando assim perda de algum dos ficheiros que compõem o visualizador.

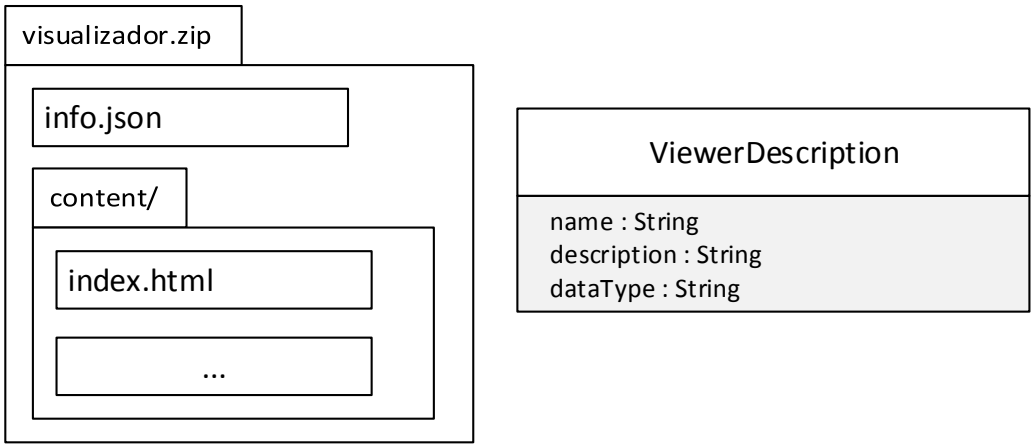


Figura 3.21: Conteúdo de um ficheiro ZIP de um visualizador

Este ficheiro tem a estrutura descrita na figura 3.21 e contem obrigatoriamente três componentes:

- O 'info.json' que é um ficheiro com as informações sobre o visualizador, informações essas descritas em JSON [70] num objeto do tipo **ViewerDescription** referido na figura 3.21;

- Uma pasta 'content' que terá todo o conteúdo que pode ser carregado para o navegador de Internet;

- Um ficheiro 'index.html' que é o ficheiro que contém apontadores para os outros ficheiros (de forma a serem carregados), tem ainda de obter o conteúdo do canal e parâmetros do visualizador, caso existam.

A indicação do canal e dos parâmetros a usar é feita por quem lança o visualizador no *browser* através do seu URL. Tal como descrito no diagrama da figura 3.22 os parâmetros são enviados em primeiro lugar para o servidor, que vai responder com um identificador (ID) que será passado para o visualizador. Depois disto o visualizador é lançado com os parâmetros "src" e "params" no URL informando do canal e ID dos parâmetros respetivamente. Por fim o visualizador usa esses parâmetros do URL para contactar o servidor para obter os parâmetros e dados do canal.

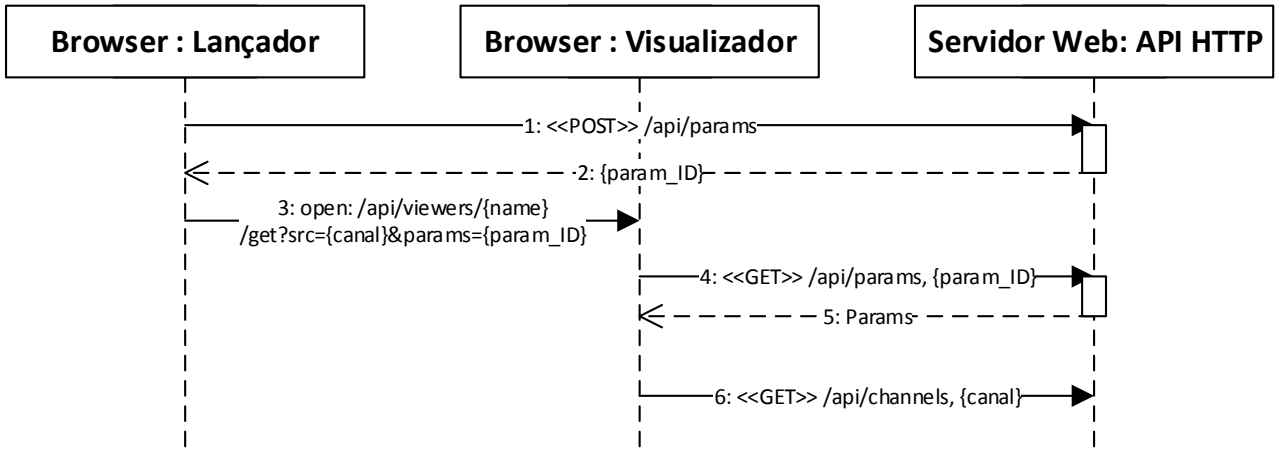


Figura 3.22: Diagrama de sequência do funcionamento da API para lançamento de um visualizador

3.5 Conclusão

Neste capítulo foi discutida a possibilidade do uso de WebRTC e apresentada uma solução. Solução essa que foi explorada e especificada ao longo de várias secções. Começando por uma biblioteca *multicast* para construir o serviço para dar possibilidade de envio de dados.

Foram também definidas soluções para entrega fiável *multicast*, gestão e partilha das chaves, autenticação dos utilizadores à sessão e manutenção da sessão.

Foi mencionado o servidor Web, que vai ser mais detalhado posteriormente. E por último, definido com detalhe as várias APIs que devem ser apresentadas sobre HTTP ao *browser*. E ainda o conceito de *plugins* e visualizadores para estender as capacidades e dar a possibilidade de definir componentes reutilizáveis.

Capítulo 4

Desenvolvimento

O desenvolvimento do trabalho apresentado foi feito de forma similar ao que foi estruturado no seu desenho. Tendo apenas o servidor HTTP e a API fundindo-se um pouco devido ao desenvolvimento de uma afetar a outra. Resultado numa estrutura semelhante à figura 4.1.

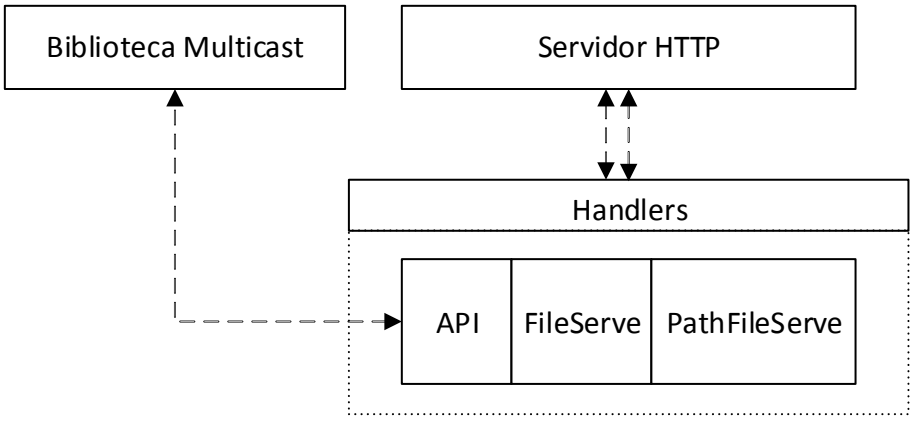


Figura 4.1: Aspecto geral de alto nível dos componentes desenvolvidos

4.1 Biblioteca *multicast*

A biblioteca *multicast*, feita inteiramente em Java tem de lidar com o envio e receção de *multicast* e ainda o processamento dos pacotes enviados/recebidos por o módulo de fiabilidade quando necessário.

Existe ainda o requisito das mensagens serem o mais pequenas possíveis em termos de tamanho, devido às limitações do tamanho dos pacotes UDP como referido em [71, 72]. Este

cuidado tornou necessário a escolha de uma boa forma de serializar os dados a enviar, focada no tamanho produzido. Desta forma foi escolhida a biblioteca Protocol Buffers.

4.1.1 Protocol Buffers

A biblioteca Protocol Buffers [73] da Google foi desenhada em 2001 para ser usada internamente para troca de mensagens entre máquinas. Em 2008 foi disponibilizada como código *open-source* sobre uma licença BSD, tornando-a uma boa escolha para projetos. A biblioteca tem algumas características de interesse que levaram à sua escolha:

- Tamanho das mensagens serializadas pequeno: segundo vários testes encontrados [74–76] é uma das que tem melhores resultados nesta área. É melhor que serialização de Java ou XML, e em comparação com outras bibliotecas semelhantes (Apache Thrift [77] ou Avro [78]) tem como vantagens uma muito boa documentação e desenvolvimento estável como referido em [74].

- Suporte de múltiplas linguagens: as mensagens serializadas numa linguagem de programação (por exemplo, em Java) podem ser desserializadas noutra (por exemplo, C++). Isto torna possível no futuro implementar o protocolo que corra sobre estas mensagens noutra linguagem de programação (por exemplo Cocoa) abrindo a compatibilidade com novas plataformas (por exemplo iOS).

Esta biblioteca gera e faz *parse* de mensagens que são imutáveis, isto é, não permitem modificação do conteúdo do objeto. Isto serve para garantir que todos os campos requeridos são garantidos e os dados não sofrem qualquer alteração. Para gerar estas mensagem existem **Builders** (um para cada tipo de mensagem) que são mutáveis.

As mensagens nesta biblioteca são primeiro definidas numa linguagem própria e posteriormente compiladas para a linguagem de destino (por exemplo Java). Com o código gerado é possível criar e fazer *parse* de mensagens.

Todas as mensagens enviadas pela biblioteca desenvolvida foram serializadas por esta biblioteca. Referências a mensagens enviadas nesta secção serão apresentadas pela definição feita para esta biblioteca, melhorando assim a legibilidade e deixando referência futura.

4.1.2 Módulos

Para implementar a fiabilidade e outros componentes dividiu-se a estrutura em módulos, que funcionam de forma singular e independente. Os módulos têm um ciclo de vida independente para processar mensagens que enviam de uns para outros.

Os módulos formam entre si uma *stack*, isto é, um conjunto de módulos em que as mensagens passam de uns para os outros e em cada um podem sofrer alterações. Por questões de simplicidade têm-se que um módulo encaixa noutro, ou seja, o que um módulo escreve é lido por outro. Sendo assim mais fácil o envio de dados entre eles.

As mensagens enviadas são definidas como eventos, um evento tem um sentido, ascendente ou descendente, na *stack* que os módulos formam e um objeto extra. São portanto definidos conforme o bloco de código em 4.1.

```
abstract class Event {
   public static final int WAY_DOWN=0;
   public static final int WAY_UP=1;

   private int way;
   private Object obj;

   public int getWay();
   public void setWay(int way);
   public Object getObj();
   public void setObj(Object obj);
}
```

Bloco de Código 4.1: Classe abstracta Event

Para passar os objetos **Event** entre módulos, cada módulo tem uma **BlockingQueue** que permite a um módulo ler objetos enquanto outro escreve. Dado cada módulo ser independente e realizar ações que podem tomar tempo de execução variável, instância-se cada um numa *thread*.

Assim sendo, a classe **Module** será base para todos os módulos e está representada no diagrama da figura 4.2. Um módulo guarda um apontador para o módulo que tem acima de si e outro para o abaixo de si, podendo assim facilmente enviar para a **Queue** do módulo acima ou abaixo.

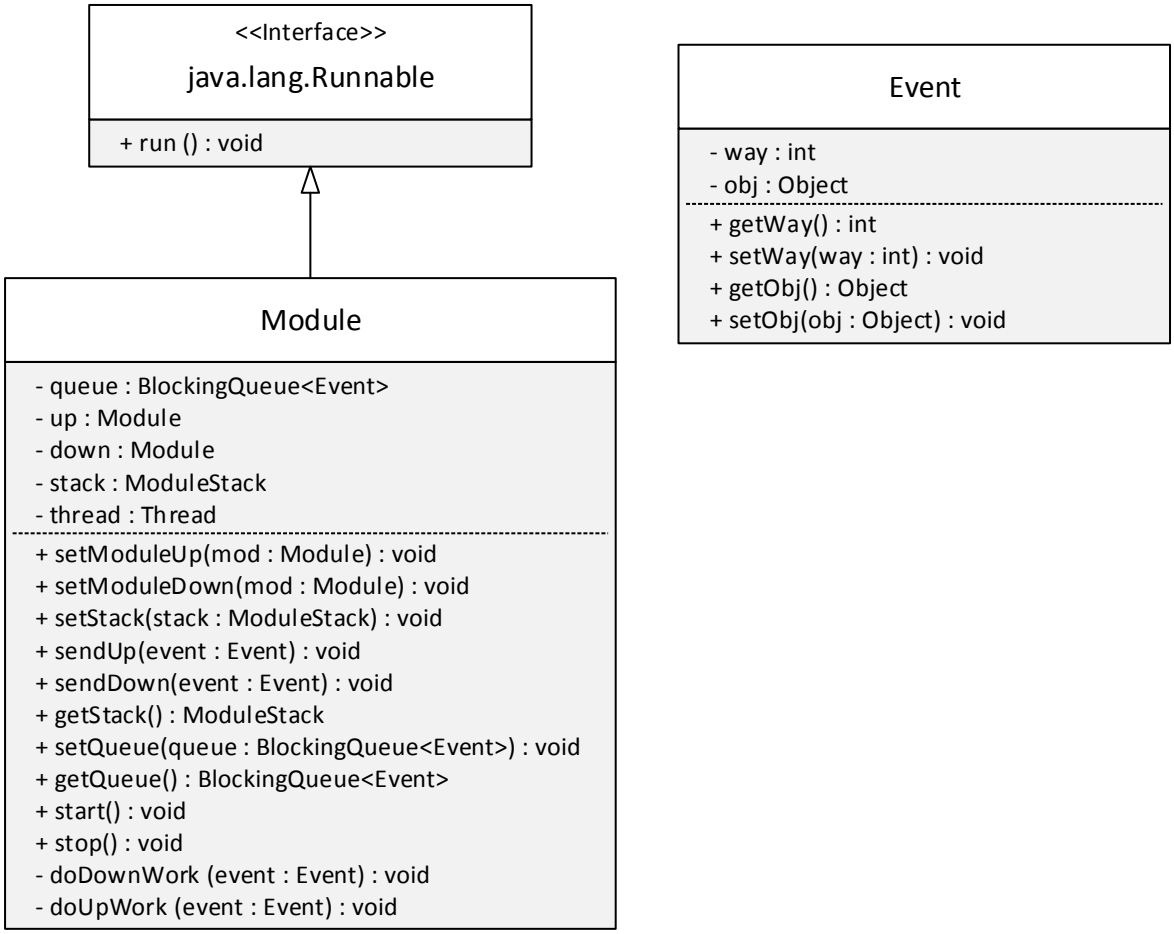


Figura 4.2: Definição da classe Module e Event

4.1.2.1 Escritores UDP

Um dos tipos de **Module** criados foi para escrever UDP (*unicast* ou *multicast*). Para abstrair e auxiliar na serialização dos dados de uma mensagem para bytes pela biblioteca Protocol Buffers foi criado a classe **TypeBuilder**, definida em 4.2. Esta classe tem internamente referência a um objeto da classe **ChannelSec** (definido no diagrama de classes 4.5), esta classe serve de ajudante na cifragem e decifragem de dados quando necessário.

O funcionamento do módulo UDP consiste em:

1. Remover da **Queue** um **Event**;
2. Obter o **Object** do **Event** e fazer *cast* para **Package**, uma classe auxiliar usada ao longo dos módulos para interagir com o pacote a enviar;

3. Usar o **TypeBuilder** para converter para um byte array;

4. Enviar por UDP para o endereço do canal;

```
public abstract class TypeBuilder<E> {
    protected ChannelSec channelSec;

    public TypeBuilder() {}
    public TypeBuilder(ChannelSec channel) {
      this.channelSec = channel;
    }

    public abstract E parseFromBytes(byte[] bytes);
    public abstract byte[] parseToBytes(E obj);
}
```

Bloco de Código 4.2: Classe TypeBuilder para auxiliar na serialização no módulo para escrita UDP

4.1.2.2 Leitores UDP

Ler de um socket *multicast* é também um módulo, sendo que o seu modo de operação é bastante simples:

- Ler do *socket multicast*;
- Fazer *parse* com o **TypeBuilder** definido;
- Criar um objeto **Package** com a mensagem recebida;
- Criar um **Event** com o objeto **Package** e enviar para o módulo acima;

As mensagens que circulam nos canais de envio de dados são **ChannelPackages**, definidos no bloco 4.3.

4.1.2.3 Fiabilidade

O módulo de fiabilidade é um dos mais complexos e importantes. Tem dois funcionamentos distintos se os pacotes forem no sentido ascendente ou descendente. Estas diferenças existem nas

```
message ChannelPackage{
   enum Type{
      MSG_DATA = 1;
      MSG_NACK_RESPONSE = 2;
   }
   required Type type = 1;  // package type
   required int32 sender_id = 2; // ID of the sender
   required int32 seq_id = 3; // package sequence number
   optional bytes data = 4; // package data
}
```

Bloco de Código 4.3: Mensagem do ProtocolBuffers ChannelPackage

funções **doUpWork(Event e)** e **doDownWork(Event e)**.

Do ponto de vista de um emissor o que o módulo faz é, caso o pacote precise de fiabilidade (por defeito todos precisam): adiciona o número de sequência, adiciona na fila de pacote enviados e envia para baixo.

O módulo tem ainda o trabalho de lidar com os pacotes do tipo NACK. Estes são pacotes que como referido em 3.2.4 informam da perda de um ou mais pacotes e o emissor deve reenviar os pacotes pedidos. Este trabalho é feito ao processar o pacote com sentido ascendente, e portanto na função **doUpWork(Event e)**. É nesta função que para quem recebe é realizado grande parte do trabalho do módulo, tem de começar por verificar se o pacote requer fiabilidade e se é do tipo dados, em caso afirmativo é processado conforme mostra o bloco de código em 4.4.

4.1.2.4 Recuperação dos pacotes

Para recuperação dos pacotes são enviados NACKs, este processo é iniciado aquando da deteção da perda de um pacote e é feito um cálculo simples de quais os pacotes em falta entre o recebido e o esperado, tendo em conta os que já foram recebidos pelo meio. A informação dos pacotes em falta é enviada para o **NACK_Sender**. Esta classe tem como única função enviar NACKs (por UDP *unicast*) conforme especificado na figura 3.8.

```
1  /**
2  * Returns if the package should be sent up
3  */
4  protected boolean processPackageFiability(Package pack) {
5     if (expected_seq_num == -1) {
6        // first package received
7        expected_seq_num = pack.getSequenceNumber();
8     }
9     if (pack.getSequenceNumber() == expected_seq_num) { // as expected
10       expected_seq_num++; dispatchReceiverQueue();
11       dismissNACK(pack.getSequenceNumber()); // dismisses the nack request to
             this package
12       return true;
13    } else if (pack.getSequenceNumber() < expected_seq_num) { // received after
          a timeout occurred for that package
14       return false;
15    } else { // pack.getSequenceNumber() > expected_seq_num
16       if (receivedPackages.containsKey(pack.getSequenceNumber()))
17          return false;// already received
18       sendNACK(expected_seq_num, pack.getSequenceNumber());
19       receivedPackages.put(pack.getSequenceNumber(), pack);
20       dismissNACK(pack.getSequenceNumber()); // dismisses the nack request to
             this package
21       return false;
22    }
23 }
```

Bloco de Código 4.4: processPackageFiability(Package pack)

4.1.3 ModuloStack

Para gerir todos os módulos ligados entre si formando uma *stack*, cada uma representa um canal, sendo que numa sessão podem existir múltiplos canais, existe o **ModuleStack** que contém e gere as *stacks* de módulos. Tem portanto como funções:

- Criar *stacks* de módulos para canais: aquando da criação de um canal criar todos os módulos necessários para o mesmo, tendo em conta se é recetor ou emissor e as necessidades de fiabilidade;

- Criar a *stack* de módulos do canal para controlo da sessão: esta *stack* é especial, existe apenas no gestor da sessão;
- Gerir *stacks* existentes: possibilidade de eliminar, ler ou escrever nos canais que as *stacks* suportam;

Assim sendo esta tem a definição apresentada na figura 4.3 em que **ChannelInfo** (representada em 4.5) é uma mensagem da biblioteca Protocol Buffers.

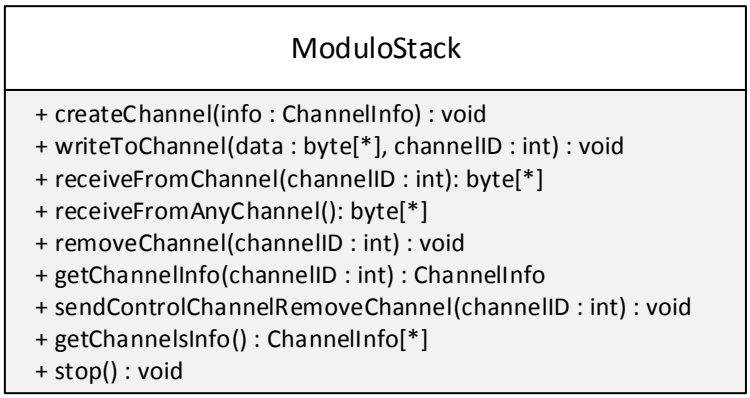


Figura 4.3: Classe ModuloStack

4.1.4 Servidores TCP

Apesar da maior parte da comunicação realizada por esta biblioteca ser em UDP existem componentes que assentam sobre TCP:

- Serviço de autenticação dos utilizadores;
- Serviço de pedido de autorização para criar um canal;

Esta escolha foi tida em conta o facto de a quantidade de ligações nestas operações ser reduzida.

4.1.4.1 Serviço de autenticação

O serviço de autenticação é divulgado nas mensagens de informação da sessão, o endereço e a porta. Os clientes ao se ligarem recebem a chave pública e aceitam ou não esta, validam a chave de acordo com a *fingerprint*. Após a validação autenticam-se seguindo o protocolo definido em 3.2.3.

```
message ChannelInfo{
   enum Type{
      CONTROL = 0;
      DATA_STREAM = 1;
      DATA_BLOCK = 2;
   }
   required Type type = 1 [default = DATA_STREAM];

   enum FiabilityType{
      NONE = 0;
      TOTAL = 1;
      OPTIONAL = 2;
   }
   required FiabilityType fiabilityType = 2 [default = TOTAL];

   required string MultiCastAddress = 3; // IP Address being used by the Channel
   required int32 MulticastPort = 4; // Port being used by the Channel to the
       Multicast
   required string Address = 5; // IP Address of the Channel
   required int32 Port = 6; // Port being used by the Channel
   required int32 senderID = 7; // ID of the Sender
   optional int32 channelID = 8; // ID of the Channel
   optional string dataType = 9; // the data type, eg: audio/mp3
   optional string description = 10; // description for the Channel
}
```

Bloco de Código 4.5: mensagem ChannelInfo

Para isto foi criado um servidor TCP na classe **AuthServer**. Cada pedido é atendido numa *thread* à parte por um objeto da classe **AuthRequestHandler**. Este modo de funcionamento tem como problema a possibilidade de abrir conexões ao servidor de autenticação até bloquear o servidor se não forem tomados cuidados.

4.1.4.2 Serviço de autorização para criar ou remover um canal

Este serviço é semelhante ao de autenticação do canal, mas o protocolo seguido é diferente, de acordo com o estabelecido em 3.2.5. O protocolo está representado no diagrama 4.4, consiste simplesmente numa mensagem inicial do cliente para o gestor da sessão cifrada com a chave da

sessão a informar o que pretende e com a posterior resposta deste. Caso seja requerido um novo canal e seja autorizado, as informações do novo canal são enviadas para o canal de controlo.

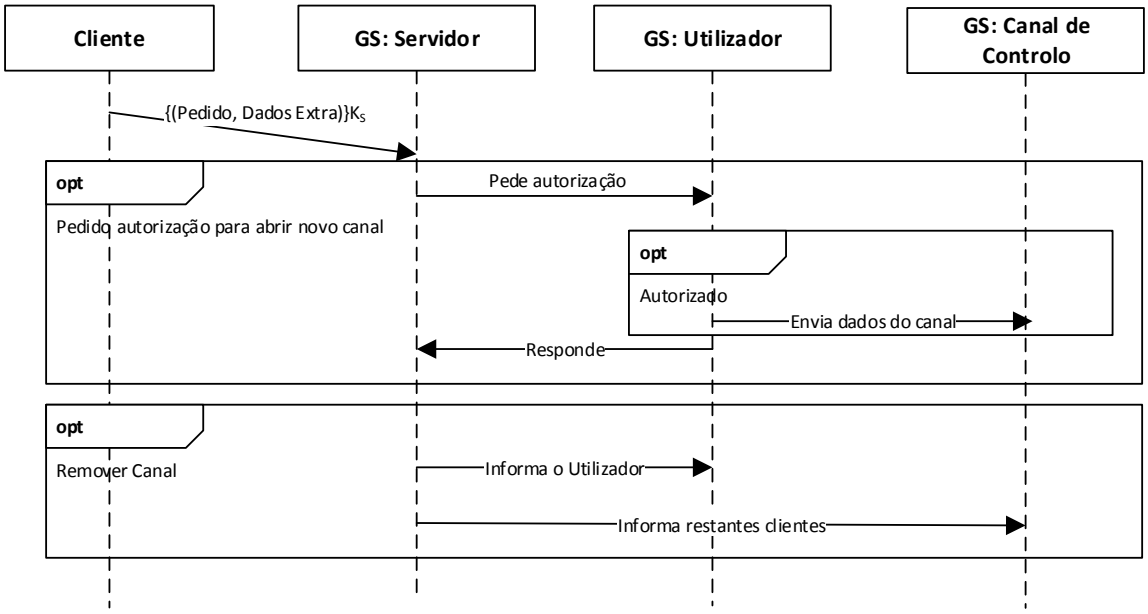


Figura 4.4: Diagrama de sequência do pedido de criação de um novo canal ou informação da remoção de um existente

As mensagens trocadas são do tipo **ProtocolMessage** definida em 4.6 dentro de mensagens **ProtocolEncriptedMessage** definidas em 4.7. O conteúdo da mensagem **ProtocolEncriptedMessage** é o resultado da mensagem **ProtocolMessage** serializada e cifrada. Isto é feito para mais fácil leitura do *socket* TCP.

```
message ProtocolMessage{
   enum Type{
      CHANNEL_REQUEST = 0;
      CHANNEL_OKEY = 1; // Accepted channel information
      CHANNEL_DENIED = 2; // Denied channel
      CHANNEL_REMOVE = 3; // informs that a channel will be removed
   }
   required Type type = 1;
   required bytes msg = 2;
   required int32 senderID = 3;
}
```

Bloco de Código 4.6: Mensagem ProtocolMessage

```
// message to send encripted messages
message ProtocolEncriptedMessage{
   required bytes message = 1; // the message encripted
}
```

Bloco de Código 4.7: Mensagem ProtocolEncriptedMessage

4.1.5 Repositório de chaves

Para criar, guardar e manter as chaves durante toda a sessão foi criado um repositório de chaves, que tem como funções guardar as chaves pública do gestor da sessão e a privada no caso deste, a chave de autenticação no caso do gestor da sessão, a chave simétrica que cifra todos os pacotes enviados da sessão conforme referido em 3.2.3.3 e obter classes para auxiliar cifrar e decifrar pacotes enviados ou recebidos respetivamente.

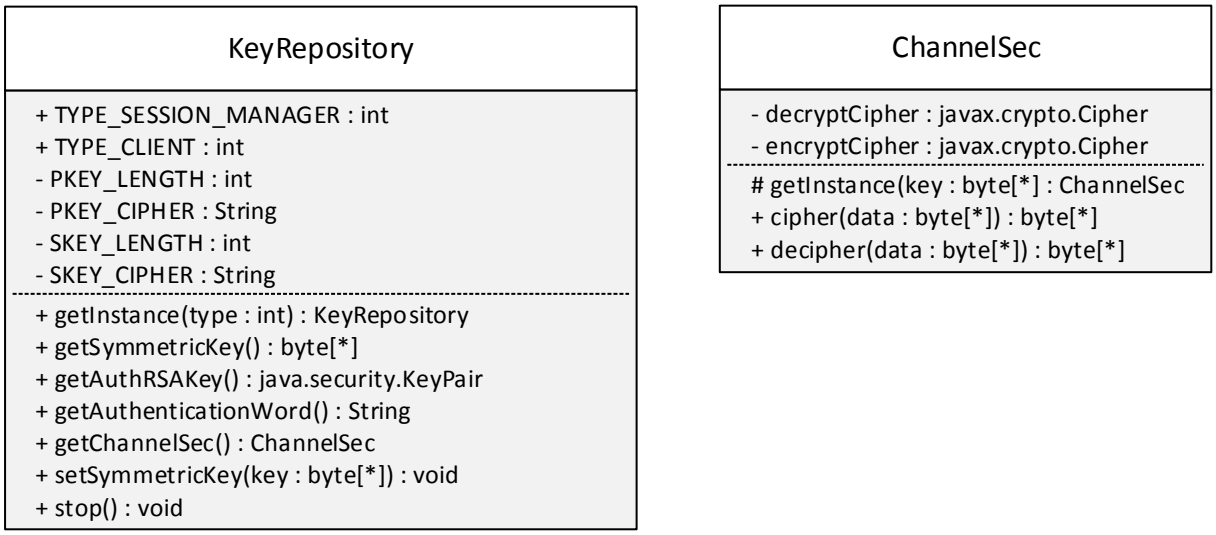


Figura 4.5: Classes KeyRepository e ChannelSec

Este objeto definido na figura 4.5 é usado no **ModuloStack** para obter os ajudantes de uso para cifragem e decifragem nos módulos de receção e envio de pacotes.

4.1.6 Serviço de anúncio e descoberta de sessão

Este serviço é iniciado aquando o início de uma nova sessão e "vive" independentemente de tudo o resto, fazendo sentido ser portanto uma *thread* independente. O funcionamento tal como definido em 3.2.2 consiste em primeiramente abrir uma *socket multicast* para um canal predefinido e

enviar periodicamente mensagens previamente construidas a anunciar a sessão conforme definido.

4.1.7 Visão externa da biblioteca

Do ponto de vista externo a biblioteca tem de dar a possibilidade de criar uma sessão ou ligar-se a uma e ter um serviço de pesquisa de sessões. Tem de após a ligação a uma sessão, ter forma de obter a lista de canais, com informações sobre os mesmos, obter e enviar dados nestes e criar novos canais.

Deste modo a visão sobre a biblioteca pode ser dividida em 3 níveis cada um inclui o anterior: cliente, em que só recebe dados e pode pedir para criar um canal e transmitir, transmissor, quando tem permissão para transmitir em um ou mais canais e gestor da sessão, que só quem cria tem esta visão. Assim sendo as permissões para cada nível estão descritas na tabela 4.1.

Permissão	Gestor da Sessão	Transmissor	Utilizador
Procurar sessões	X	X	X
Ligar a sessão	X	X	X
Obter lista de canais	X	X	X
Receber de um canal	X	X	X
Pedir um canal para transmitir	X	X	X
Transmitir para um canal	X	X	
Obter lista de utilizadores	X		
Adicionar gestor de pedidos de autorização para transmitir	X		
Adicionar gestor de pedidos de autenticação	X		

Tabela 4.1: Permissões por grupo da biblioteca *multicast*

Foram assim desenhadas um conjunto de interfaces consoante as permissões e implementadas por um objeto que abstraí a comunicação com o módulo **Stack** que gere os canais, como mostra a figura 4.6.

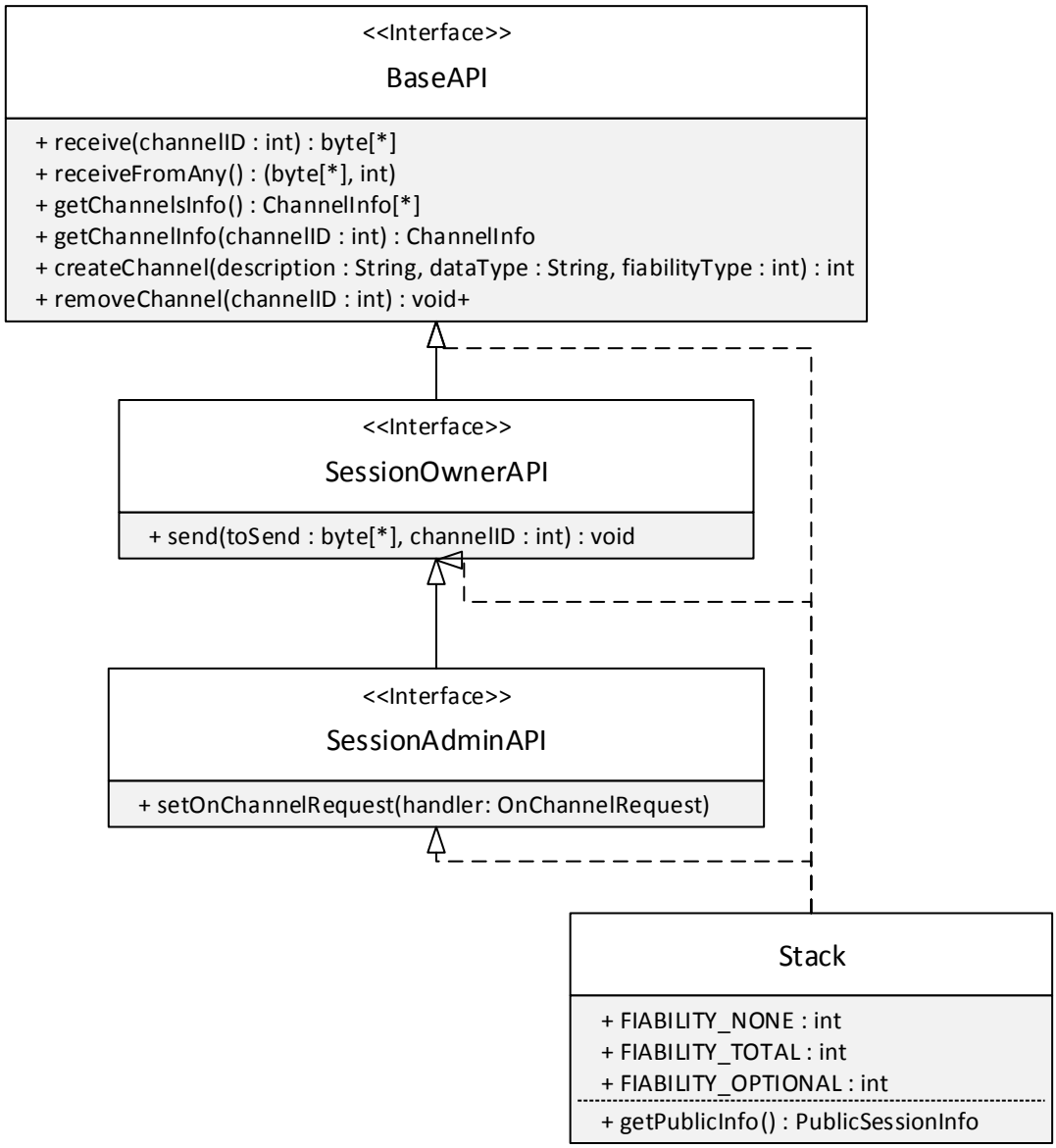


Figura 4.6: Interfaces da API da biblioteca *multicast*

4.1.8 Visão interna da biblioteca

Do ponto de vista interno a biblioteca apresenta-se conforme a figura 4.7. O que é exposto para fora é a **Stack** e o **KeyRepository**. Este segundo dá a possibilidade de configuração do dicionário de palavras a usar para a *fingerprint* para validação da chave pública e a chave de autenticação.

Internamente existe a **InternalStack** para gerir a comunicação da **Stack**, que expõe a biblioteca para o exterior com as várias partes internas. A **InternalStack** tem o serviço para anunciar a sessão (o **SessionDiscovery**), o servidor de autenticação (**AuthServer**) e o **ModuleStack**. Este último gere a criação de *stacks* de **Module**. Os módulos usados são: o de

Fiabilidade (que tem opções para tempo real e lida com envio e receção), o **NACK_Sender** que gere os pedidos de NACKs, o de envio UDP e receção UDP.

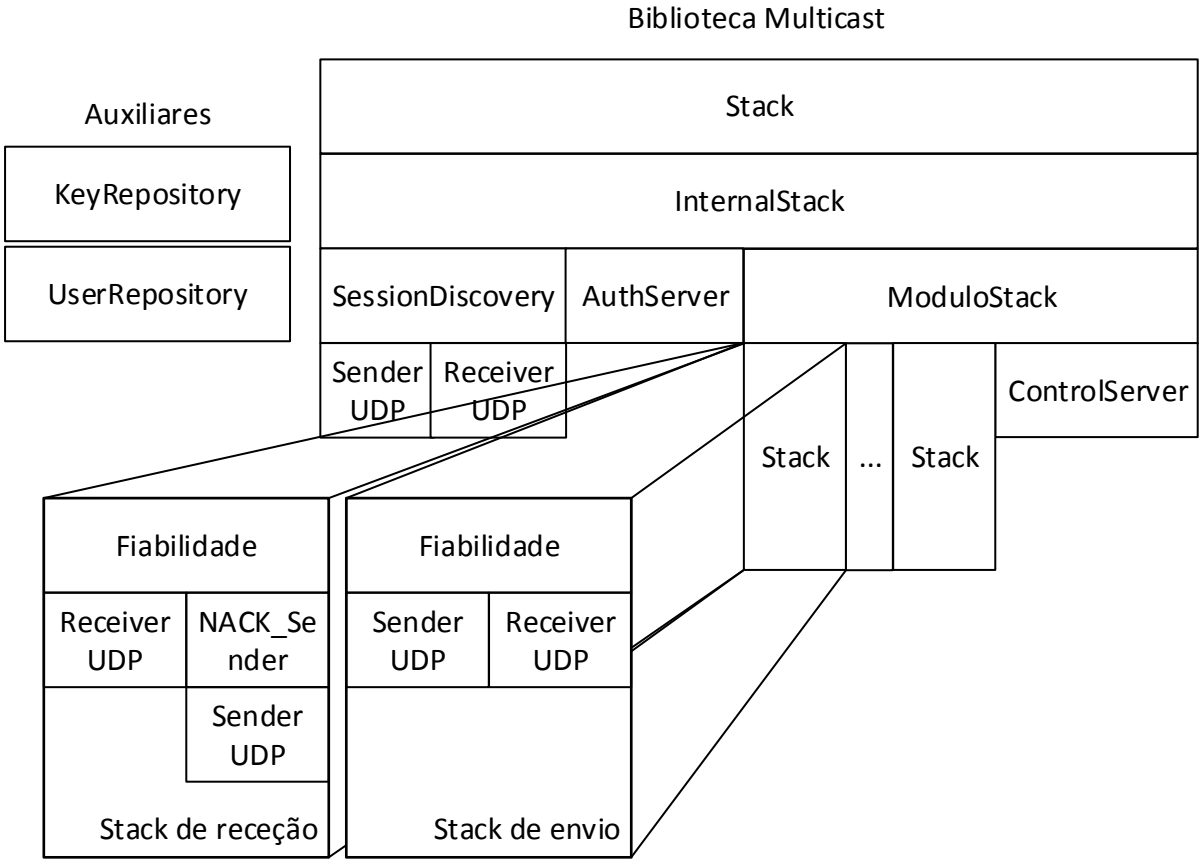


Figura 4.7: Aspeto interno da biblioteca

4.2 Servidor HTTP

O servidor HTTP foi desenvolvido com o objetivo de ser o mais simples possível, mas suportar as necessidades de servir a biblioteca sobre uma API HTTP. Como tal foi implementado para suportar apenas o estritamente necessário:

- Cabeçalhos dos pedidos HTTP: fazer *parse* dos cabeçalhos recebidos e dar possibilidade de quem responde ao pedido obter os mesmos;
- Dar capacidade para definir o código de resposta HTTP;

- Suportar HTTP 1.1 [79]: em HTTP 1.0 apenas os métodos GET, POST e HEAD são suportados segundo o Request for Comments (RFC) 1945 [80], para suportar toda a API definida em 3.4 são necessários os métodos DELETE e PUT além do GET e POST;
- Definir cabeçalhos na resposta: conseguindo assim manter estado através de *cookies*, por exemplo;
- Definir o tipo de dados da resposta;
- Suportar escrever livremente na resposta do pedido HTTP;

4.2.1 *Parsing* dos pedidos

Para facilitar as interações com os pedidos e respostas HTTP foram criadas duas abstrações, o **Request** e o **Response**. O **Request** com funções para lidar com o input do pedido, fazer *parse*, construir estruturas para facilitar o acesso aos cabeçalhos recebidos e conseguir ler o corpo do pedido em caso de necessidade.

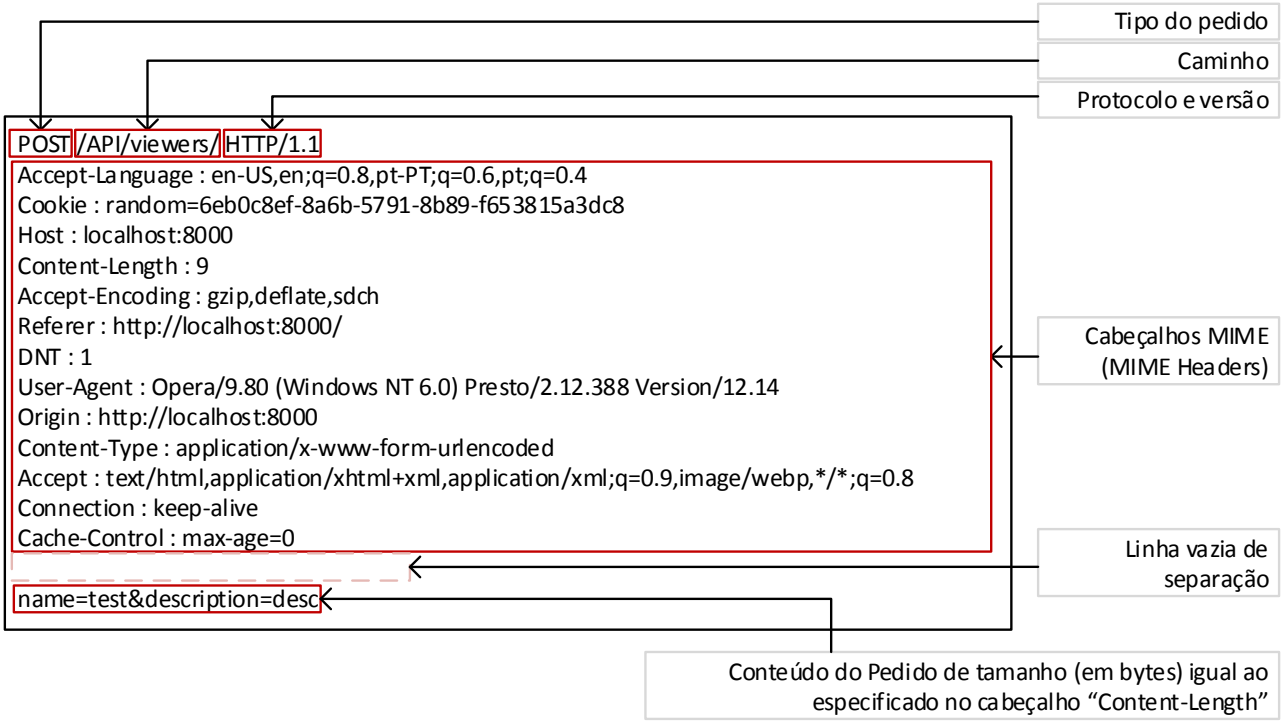


Figura 4.8: Formato de um pedido HTTP

Um pedido HTTP tem a estrutura apresentada na figura 4.8 conforme definido no RFC 1945 [80] . Assim sendo é feito uma leitura linha a linha até encontrar-se dois 'LF+CR', a partir desse ponto começa o corpo do pedido.

4.2.1.1 Corpo do pedido

O corpo do pedido não é feito *parse*, este tem contexto, depende do tipo de dados que for enviado. Para tal faz sentido disponibilizar acesso ao *input stream* diretamente e deixar quem atende o pedido decidir como ler os dados.

Apesar desta decisão foi feito um ajudante para o caso dos formulários HTML, o **FormRequest**. Existem dois tipos de formulários distintos, o **application/x-www-form-urlencoded** e o **multipart/form-data** segundo o standard do W3C em [81] . No primeiro o conteúdo é escapado e enviado para o servidor com os pares nome do controlo e valor separados por "=" estes pares de nome do controlo e valor separados entre si por "&".

O segundo é necessário para formulários com ficheiros ou texto com caracteres não ASCII será útil para o envio de *viewers* para o servidor.

4.2.2 Handlers

Para tratar de um pedido HTTP foi construído o conceito de **Handlers**, que são objetos que implementam a interface **HandlerInterface** definida na figura 4.9. Aquando da definição do servidor HTTP são adicionados para responder aos pedidos para um dado caminho.

Estes *handlers* têm simplesmente um ponto de entrada que é chamado quando um pedido para o caminho que eles tratam for feito, e vivem entre múltiplos pedidos diferentes (podendo assim guardar estado). É necessário cuidado, pois o servidor é *multithread*, isto é, pode atender mais que um pedido de cada vez. Se o código da função chamada alterar o estado interno do **Handler** pode resultar em problemas por várias *threads* estarem a fazer alterações em simultâneo.

Um conjunto de **Handlers** básicos foram definidos para ajudar no funcionamento base do servidor, por exemplo para servir ficheiros singulares de forma estática, servir ficheiros de baixo de um caminho como se de um caminho local se tratasse, ou responder a notificações.

4.2.3 Construção do servidor

A criação do servidor é feita através do padrão Builder definido na classe **Server.Configuration**.

Assim torna-se flexível a adição e remoção de **Handlers** consoante as necessidades. Tem como ponto problemático o facto de em tempo de execução não ser possível adicionar ou remover

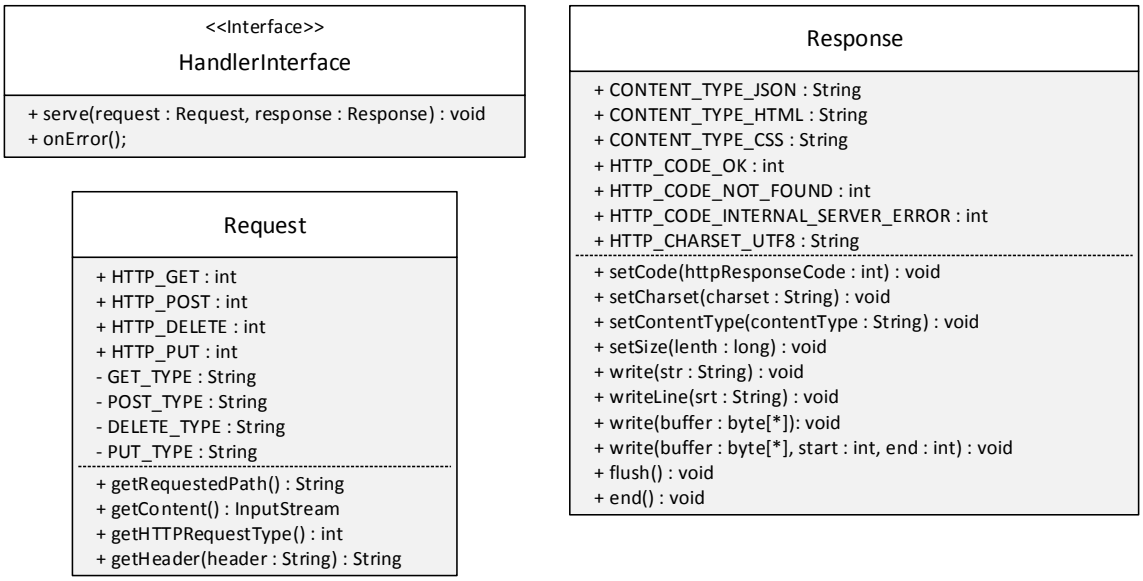


Figura 4.9: A interface HandlerInterface e as classes Request e Response

Handlers. Isto pode ser acrescentado ao **Server** posteriormente pela definição de um **Handler** que permita adicionar **Handlers** e encaminhe para estes os pedidos.

Um exemplo de construção do servidor pode ser encontrado no bloco de código 4.8.

```
Server s = new Server.Configuration()
   .setIP(8000)
   .serve("/", new FileServe("index.html",
       pt.esoares.utils.Files.GENERIC_CONTENT_TYPE_RESOLVER))
   .buildServer();

s.start(); // inicia o servidor
Thread.sleep(10000 * 10 * 5); // deixar o servidor correr
s.stop(); // para o servidor
```

Bloco de Código 4.8: Exemplo de criação de um Server

4.3 API HTTP

A API sobre HTTP foi implementada muito próximo do servidor Web pois o funcionamento desta foi moldada pela forma de interação do pedido HTTP.

4.3.1 Serialização em JavaScript Object Notation (JSON)

Para facilitar o envio de informações para o *browser* e poderem ser manipuladas foi optado por os usar objetos serializados em JSON [70]. Para tal foi utilizada a biblioteca GSON [82] da Google, que tem a licença Apache 2.0 [83] , sendo assim possível incluir e usar neste trabalho. Um dos pontos importantes para a escolha desta biblioteca foi o facto de oficialmente suportar Android [84].

Esta biblioteca tem como particularidades ser fácil para serializar objetos e definir a exposição de atributos com anotações Java, tem ainda a possibilidade de definir conversores para objetos em que não se possa alterar o código fonte (ou seja preferível não o fazer). Podendo assim sem qualquer alteração na biblioteca *multicast* construída expor representações personalizadas dos objetos gerados por ela por exemplo, informações das sessões existentes.

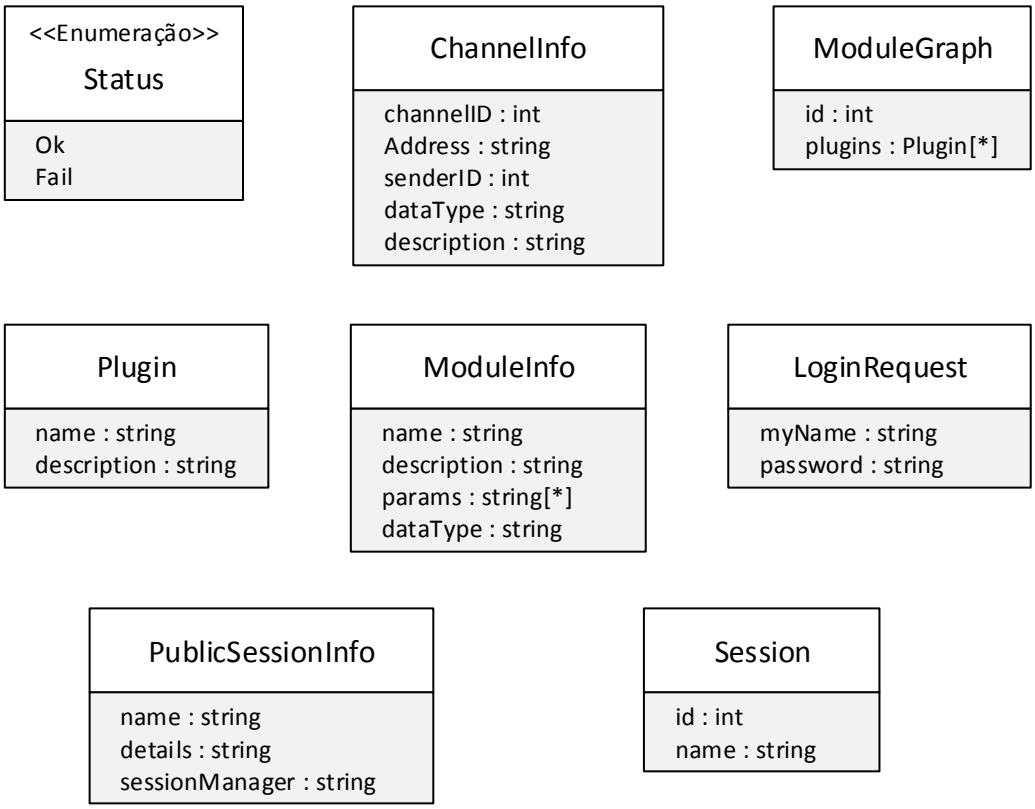


Figura 4.10: Classes JSON da API HTTP

Assim sendo os seguintes objetos JSON expostos com as várias informações expostas pela API estão definidos na figura 4.10.

4.3.2 HTTP Handler

A API foi construída no servidor como um **HTTPHandler**, o que faz com que a cada pedido para um dado caminho definido seja encaminhado para o **Handler** construído.

Este *handler* fica registado no caminho base "/api/*", podendo gerir assim internamente os sub-caminhos possíveis. Assim sendo, foi mapeada toda a API definida em 3.4 sobre este caminho.

O *handler* construído internamente tem um conjunto de *handlers*, um para cada sub-caminho da API e encaminha cada pedido para o *handler* responsável por o sub-caminho. Cada um destes *handlers* é instanciado a cada pedido e portanto foram criados um conjunto de classes auxiliares que seguem o padrão *Singleton* para assim manter estado entre vários pedidos.

As classes para responder ao pedido estão definidas na figura 4.11, para auxiliar com a manutenção do estado foram definidas as classes da figura 4.12.

4.3.2.1 Descoberta e criação de sessões

A API para descoberta e criação de sessões é composta por um conjunto de estados descritos na figura 4.13, expostos pela API definida em 4.11.

4.3.2.2 Canais

Os canais são o mais simples, pois com ajuda da referência da **Stack** guardada na classe de ajuda **Session** descrita na figura 4.12 basta pedir os dados do canal diretamente a esta. Não é necessário informação extra, tornando assim simples obter dados e enviar para o *browser* a cada pedido.

4.3.2.3 Plugins

Os *plugins* são dos componentes mais complexos da API desenhada. Estes têm um ciclo de vida independente dos pedido HTTP, mas a sua criação, interação e acesso à saída de dados é realizada a partir desta.

Conforme descrito na secção 3.4.5 existem diversos tipos de *plugins* que devem ser tidos em

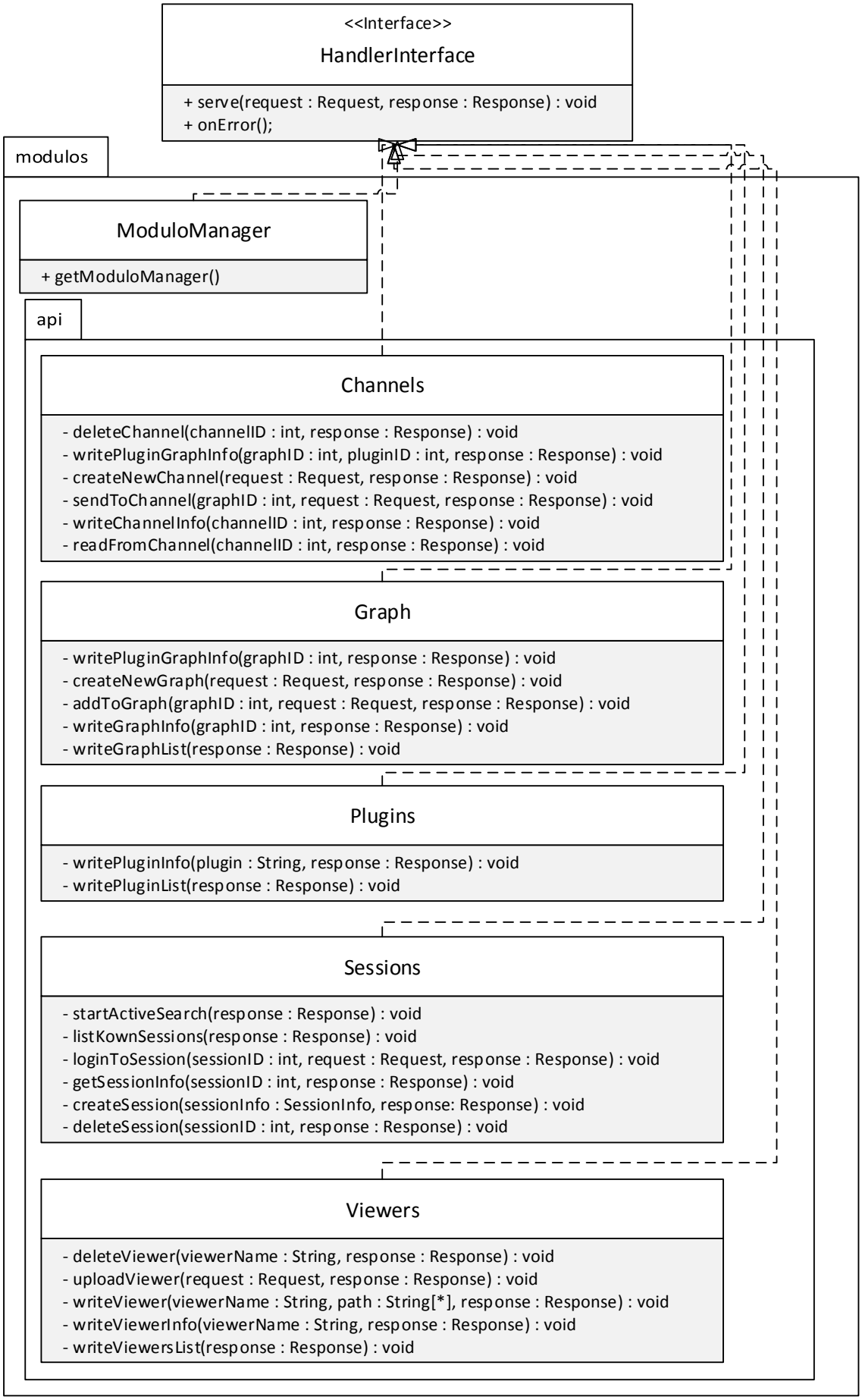


Figura 4.11: Classes que tratam dos pedidos à API HTTP

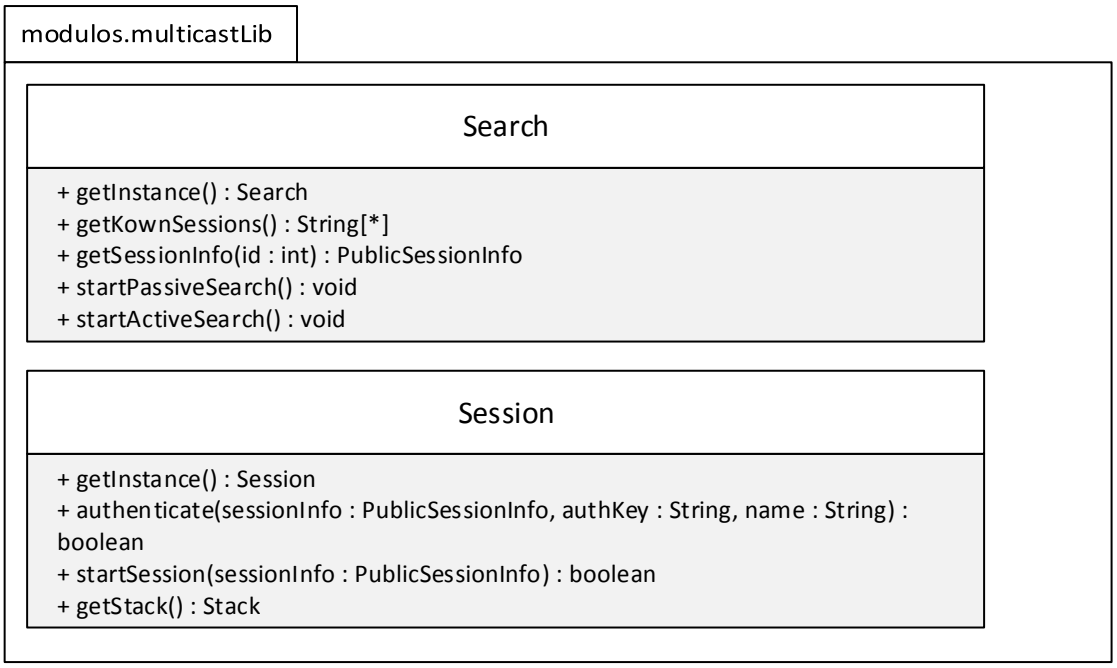


Figura 4.12: Classes ajudantes entre as respostas a pedidos HTTP e a interação com a API *multicast*

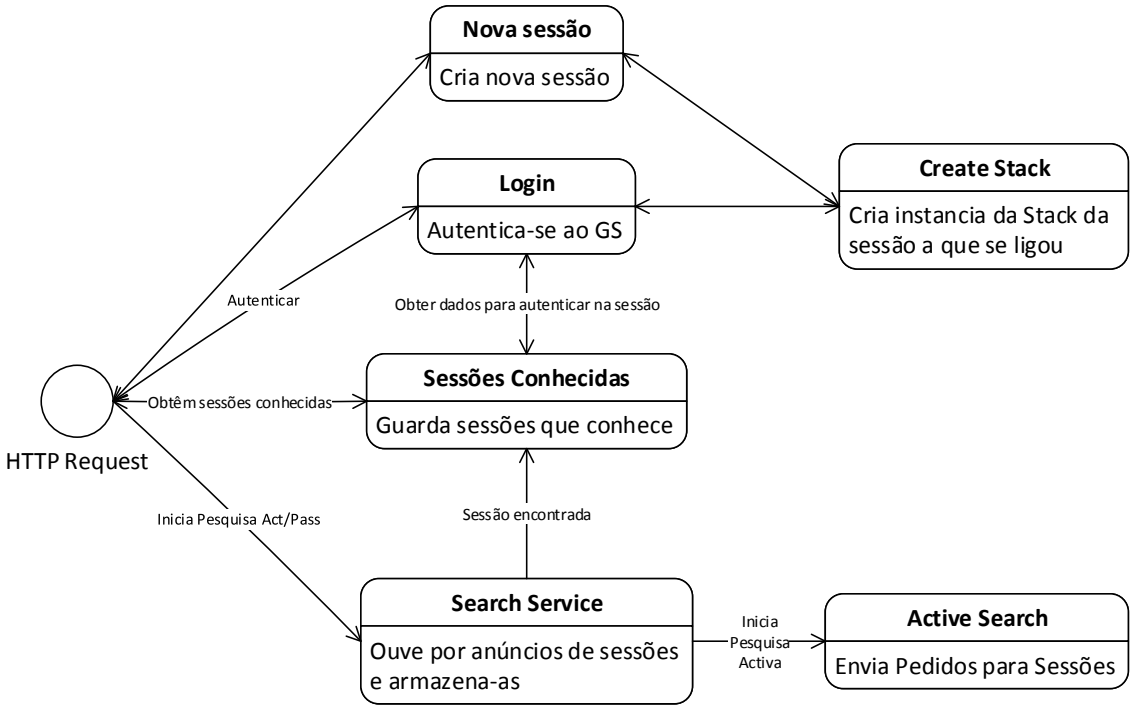


Figura 4.13: Estados da descoberta e criação de sessões

Este diagrama representa os vários elementos que um pedido HTTP na API de descoberta e criação de sessões. Mostra os vários componentes existentes e como estes interagem entre si mediante o pedido efetuado.

conta. A formação dos grafos deve respeitar os tipos dos *plugins*. Além disso existe ainda o problema de como adicionar *plugins* construídos por terceiros.

Para manter os *plugins* existentes e interação com estes (adicionar ou remover) foi criada um repositório, o **ModuleRepository**. Um objeto que guarda informações destes, permite obter instâncias deles e permite adicionar novos. Assim quem use esta biblioteca e queira adicionar *plugins* só tem de na base de aplicação em Java, após importar a biblioteca, obter o repositório e adicionar o **Plugin**, como demonstrado no bloco de código 4.9.

```
public void Start(){ // in the Java part of the app where the server is started
    // ...
    ModuleRepository repo = ModuleRepository.getInstance();
    repo.addPlugin(....);
    // ...
}
```

Bloco de Código 4.9: Adicionar um **Plugin** no **ModuleRepository**

Internamente existe um gestor de grafos de *plugins* chamado **ModuleManager**, que existe entre pedidos HTTP e serve de controlador para criação e interação com grafos de *plugins* que estejam a correr.

A criação de *plugins* envolve um conjunto de passos:

1. Para cada **Plugin**:
   (a) Obter uma instância deste no **ModuleRepository**;
   (b) Dar-lhe os parâmetros recebidos no pedido HTTP;
2. Criar as ligações entre os *plugins*;

Para ligar os *plugins* foram testadas uma série de soluções:

- Envio por TCP: a fonte de dados escreve para um *socket* TCP e o leitor de dados lê de outro *socket* TCP ligado ao primeiro. Esta solução delega o *buffering* ao *buffer* dos *sockets* TCP tornando-se simples de implementar, mas tendo como problema o facto de para múltiplos consumidores dos dados de uma só fonte (em que todos têm de receber todos os dados), terem de existir um par de *sockets* (um para leitura outra escrita) por leitor;

- Envio por *multicast*: a fonte de dados escreve para um canal *multicast* e cada leitor lê do canal. Para garantir que não é enviado para a rede externa os pacotes levam um TTL de valor zero, mesmo que estes sejam transmitidos pelo dispositivo devem ser descartados no primeiro *router*. Esta forma de envio tem o problema de não haver *buffering* de dados e portanto é fácil de haver perdas se os *plugins* que estiverem a ler demorarem a processar os dados. Tem como grande vantagem a simplicidade para o emissor de dados e baixo gasto de memória;

- Envio por *Pipes*: esta é uma forma mais tradicional em UNIX [85] de comunicação entre processos. O seu funcionamento consiste num *buffer* em memória no qual um escritor coloca dados e um leitor lê, formando assim um *Pipe* a ideia de um tubo em que de um lado se coloca dados que saem no outro lado. Este sofre de problemas semelhantes à solução dos *sockets* TCP (ter de existir um *Pipe* para cada leitor), mas a entrega de dados é fiável e o *buffering* é suportado pela implementação. Implementação essa existente em Java nas classes **java.io.PipedInputStream** e **java.io.PipedOutputStream** para escrever e ler dados escritos da *Pipe* que a classe **PipedOutputStream** tem.

Após estes testes achou-se que o mais adequado seria usar uma solução de entrega de dados fiável quando o grafo precisa de fiabilidade caso contrário a escolha recai numa solução mais simples em termos de funcionamento e memória. A escolha da solução fiável acontece quando a fonte de dados ou uma das saídas precisa de fiabilidade um exemplo seria um canal *multicast* fiável como fonte de dados. Assim sendo foi escolhido usar-se *sockets* TCP quando fiabilidade é necessária e *sockets multicast* caso contrário.

Para cada um dos *plugins* foi criada uma classe abstrata base que se deve estender, dando assim um conjunto de funcionalidades necessárias (criar ligações entre *plugins* e ajuda para ler e escrever dados de entrada e saída). A classe abstrata está representada na figura 4.14, em que os métodos **getInputStream()** e **getOutputStream()** permitem de forma simples obter *streams* para ler e escrever para a saída respetivamente. O método **getInputStream()** só funciona correctamente quando o plugin tem apenas uma fonte de dados, assim garante-se que quando existe mais que uma, ele tem de explicitamente ler de um **SingleInput** do *array* **sources** a fonte de dados que quer.

O objeto **Memory** é um ajudante para escrever num **SingleOutput**, mas podem haver vários leitores a ler, havendo para tal várias implementações usando a mais adequada conforme discutido anteriormente.

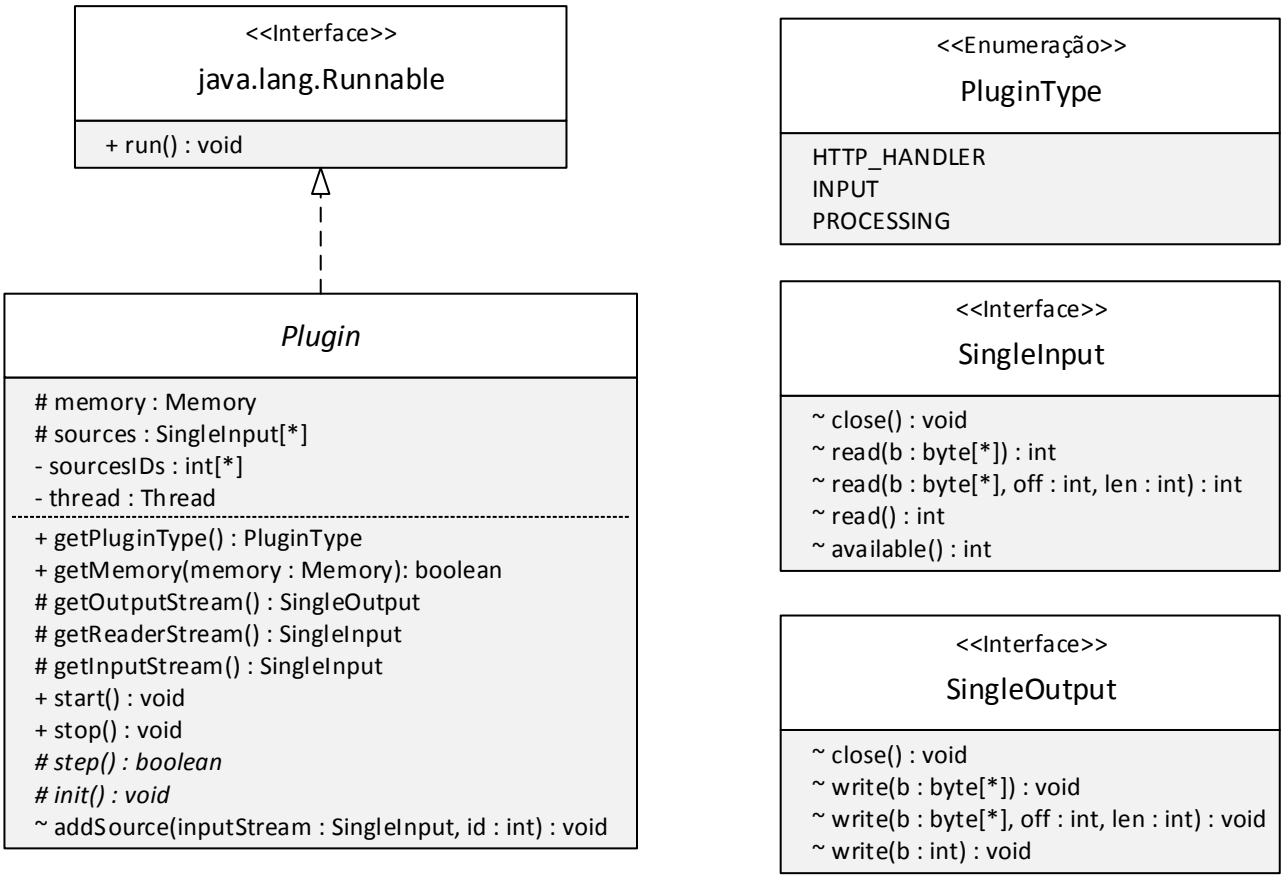


Figura 4.14: Classe abstrata Plugin

Para um *plugin* ler de um canal de dados da biblioteca *multicast* pode aceder a este da mesma forma que a API acede, ou seja, através da classe ajudante **Session** definida em 4.12. Obtendo uma referência à sessão o *plugin* pode ler diretamente de um canal.

4.3.2.4 Visualizadores

Os visualizadores para além da implementação desenhada na secção 3.4.6 foi desenvolvida uma solução para poder ter visualizadores embebidos com a aplicação Web, colocados lá quando esta é construida e distribuída e ainda assim ser possível adicionar novos pela API HTTP.

Para tal foi desenhado no repositório que contém os *plugins* o suporte para carregar os visualizadores. O funcionamento consiste em ter duas localizações, uma para visualizadores que vêm com a aplicação, outra para visualizadores que foram carregados pela API (chamada de pasta temporária de visualizadores).

Para gerir isto, existe uma *cache* em memória com os nomes dos visualizadores e a localização destes. Caso um novo visualizador que substitua um já existente seja carregado, é colocado na

pasta temporária de visualizadores e substitui na *cache* a referência para o visualizador. Para voltar a ter o visualizador original, basta remover o carregado e ao reiniciar a aplicação e a *cache* ao ser reconstruida vai voltar a ter a referência para o original.

Esta *cache* é implementada com um **HashMap** no **ModuleRepository** em que referencia nomes do visualizador em String com a localização deste.

4.4 Estrutura final

A estrutura final destes três componentes construidos: biblioteca *multicast*, servidor HTTP e API via HTTP podem ser vistos da forma apresentada no diagrama da figura 4.15. Em que a biblioteca *multicast* e o servidor Web são independentes e podem ser usados por si só. A componente da API é meramente um **handler** de pedidos HTTP. Apesar do código desta estar acoplado ao servidor Web, estes podem ser facilmente separados e usados independentemente.

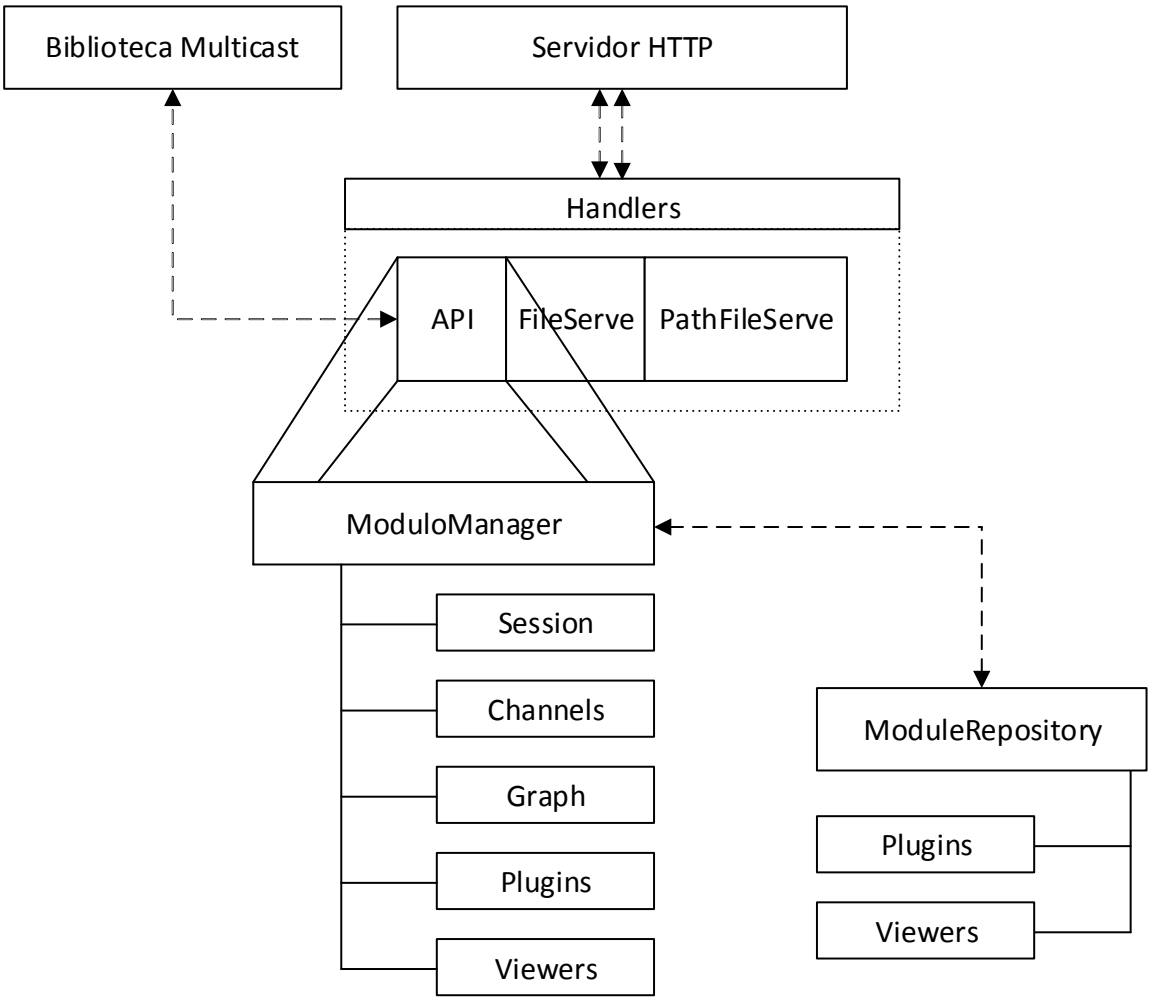


Figura 4.15: Aspecto geral dos componentes implementados

A instanciação completa para uso destes componentes segue os seguintes passos:

1. Definir a localização dos visualizadores incluídos na aplicação e a pasta para os temporários;
2. Construir o servidor através do builder **Server.Configuration** com os seguintes **Handlers**:
   - **FileServe**: para servidor o index.html;
   - **PathFileServe**: servir todos ficheiros abaixo da pasta *files* usada assim para facilitar as configurações de quais ficheiros podem ser referenciados por o index.html e para facilitar a execução em Android, mais detalhes sobre este ponto são dados na secção 4.5;
   - **ModuleManager**: responsável pela API criada. Esta acede à biblioteca *multicast*, cria e manipula visualizadores e *plugins*;
3. Colocar o servidor a correr;

O resultado é um bloco de código semelhante ao do bloco 4.10. De notar que o **FileServe** e o **PathFileServe** aceitam como argumentos a localização do item (ficheiro ou pasta) que vão servir e uma classe para resolver os tipos de ficheiros. No exemplo é usada uma por defeito incluída no servidor. Mais detalhes sobre a necessidade de existir esta opção podem ser encontrados na secção seguinte.

4.5 Execução em Android

Todo o trabalho desenvolvido foi pensado para funcionar em múltiplos sistemas operativos: Android e Windows. Dado o trabalho ter sido desenvolvido e testado ao longo do desenvolvimento em Windows era conhecido que funcionava corretamente neste. Mas para funcionar em Android vários cuidados foram tidos ao longo do desenvolvimento.

Em primeiro lugar foram usadas bibliotecas que funcionavam corretamente em ambos os sistemas operativos: a escolha da Protocol Buffers e GSON, em segundo o cuidado de usar apenas APIs que existiam em ambos os SDKs. E um último cuidado de poder injetar dependências, isto é, algo sobre o qual uma funcionalidade é depende e que é externa (por exemplo o dicionário de palavras para converter a *fingerprint*) poder ser fornecido em tempo de execução no arranque da aplicação.

```
ModuleRepository repo = ModuleRepository.getInstance();
repo.setViewersBuiltintDir(new File("files" + "viewers" + File.separator +
    "builtin"));
repo.setViewerTempDir(new File("files" + "viewers" + File.separator + "temp"));

Server s = new Server.Configuration()
   .setIP(8000) // opcional
   .serve("/", new FileServe(new File("files" + File.separator
       +"index.html").getAbsolutePath(),
       pt.esoares.utils.Files.GENERIC_CONTENT_TYPE_RESOLVER))
   .serve("/files/.*", new PathFileServe(new File("files" + File.separator
       +"files").getAbsolutePath(),
       pt.esoares.utils.Files.GENERIC_CONTENT_TYPE_RESOLVER))
   .serve("/API/.*", ModuleManager.getModuleManager()) // obter o ModuleManager
   .buildServer();
s.start();
```

Bloco de Código 4.10: Exemplo de criação de todos os componentes para execução

Com todos estes cuidados foi apenas necessário duas injeções de dependências consoante a plataforma:

- Ficheiros: os ficheiros usados (o dicionário de palavras, os visualizadores e a aplicação construida em HTML+Javascript+CSS) são referenciados de forma diferente em Android. Dado não ser possível embeber e navegar livremente pela aplicação compilada e assim ler o conteúdo embebido com esta, foi optado por criar um ZIP com estes ficheiros. Este ZIP é extraído no arranque da aplicação para uma localização que pode ser livremente lida por ela, esta localização é fornecida à biblioteca.
- Deteção do tipo de ficheiros: existem várias bibliotecas para isto (seria uma possível solução), mas em ambos SDKs existem opções para tal. Por isso optou-se por fornecer no arranque uma classe que soubesse resolver o tipo de ficheiros;

A estrutura de uma aplicação Android (pode ser visto um exemplo na figura 4.16) tem uma pasta para o código Java e outra para recursos (res): *strings*, *layouts* de interfaces, imagens e recursos diversos. Existe um particular uma pasta nos recursos de nome **raw** que permite qualquer ficheiro, e pode ser lido pela aplicação. É portanto o local ideal para adicionar o ZIP com todo o conteúdo da aplicação desenvolvida.

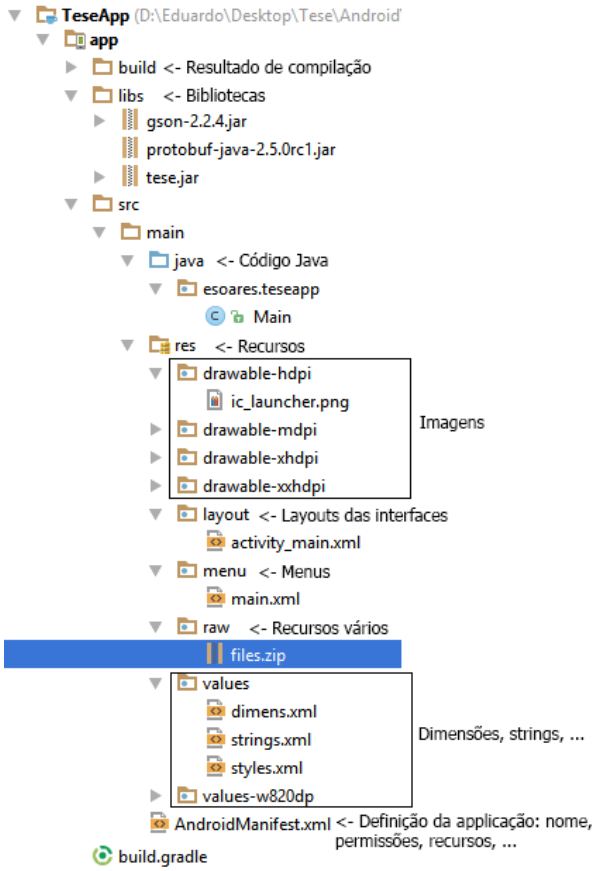


Figura 4.16: Exemplo da estrutura de uma aplicação Android

4.5.1 Serviço de suporte

Uma aplicação em Android tem uma Activity e é nesta que é apresentada a interface gráfica e é respondido aos toques nela. Um utilizador pode a qualquer momento ausentar-se desta (para por exemplo responder a uma SMS) e a sua execução pára. Isto é problemático se colocar-mos a executar o serviço que recebe os dados neste ambiente, pois podemos perder mensagens se este for parado a meio.

Por isso e de acordo com o ciclo de vida dos vários componentes possíveis [86] foi escolhido o uso de um serviço. Este executa independente do ciclo de vida da Activity e mesmo que o utilizador se ausente por momentos não corre tanto risco de ser terminado. Todos os serviços podem ser terminados a qualquer momento por o sistema operativo, mas tem muito maior prioridade para continuar a correr que uma Activity que não está visível.

Foi criado um serviço que a aplicação liga ou desliga para iniciar ou terminar todos os serviços (o servidor Web com a API e a biblioteca *multicast*).

4.5.2 Aplicação base

Para visualização da aplicação Web desenvolvida é usada uma Activity com uma **WebView** mostrando a página de entrada index.html que carrega todos os recursos. Para isto funcionar da forma mais correta e sem necessidade de qualquer alteração do código Java deve ser usada a estrutura apresentada na figura 4.17.

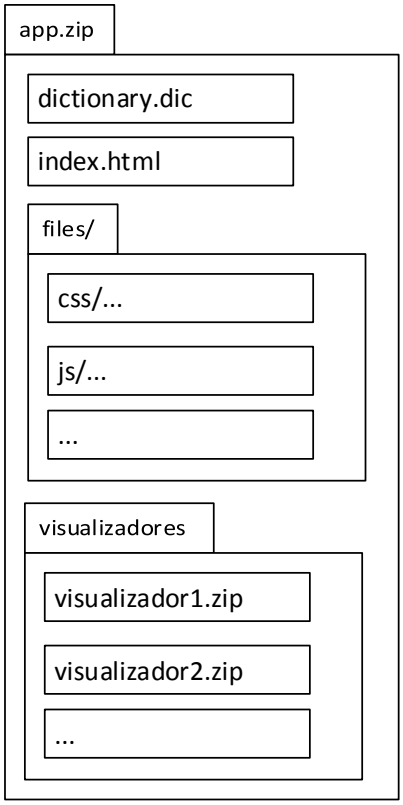


Figura 4.17: ZIP com a aplicação Web para correr em Android

Desta forma é possível o servidor resolver e servir todo o conteúdo pedido que está o caminho da pasta *files*. O uso do index.html resolve o problema de encontrar o ponto de entrada para a aplicação. Visualizadores podem também ser embebidos na pasta **visualizadores**, cada um em seu ZIP de acordo com o definido na secção 3.4.6.

O resultado é portanto uma base de uma aplicação Android com tudo o que é necessário para correr. Para adicionar a aplicação desenvolvida basta colocar num ZIP todo o conteúdo necessário, colocar na pasta **raw** e compilar a aplicação.

A aplicação base tem ainda os cuidados necessários de pedir as permissões para uso de *sockets* de rede e *multicast*, para tal declara no ficheiro **AndroidManifest.xml** na base da aplicação as permissões:

- android.permission.INTERNET: necessário para abrir *sockets* de rede;
- android.permission.CHANGE_WIFI_MULTICAST_STATE: necessário para receber sem problemas pacotes *multicast*;

4.5.3 Aplicações Nativas

Além da aplicação base resultado para dar a possibilidade de uso da biblioteca *multicast* em aplicações Web, dada a estrutura modular é possível usar a biblioteca construída diretamente numa aplicação nativa Android. Para tal basta importar a mesma para a aplicação e criar a sessão ou ligar a uma, isto convém ser feito num serviço para evitar problemas como feito na aplicação base para aplicações Web.

Além disto é ainda indispensável declarar as permissões necessárias conforme referido anteriormente.

4.6 Conclusão

Neste capitulo foram expostos detalhes do desenvolvimento dos vários componentes definidos e estruturados no capítulo 3.

Um dos componentes, a biblioteca *multicast*, foi o componente com mais complexidade. A abstração dos componentes em módulos que permitem ser colocados uns sobre os outros, formando assim uma *stack* é importante, pois torna abstrato o seu uso e manipulação, dando a possibilidade de fácil substituição de módulos entre si. Detalhes sobre a manipulação das chaves e abstração feita do seu uso foram também feitos. Terminando a secção mostrando o resultado do ponto de vista interno e externo.

O servidor HTTP foi nesta secção mais detalhado, pois foi criado consoante as necessidades de implementação da API definida. Para direcionar pedidos foi criado o conceito de *handlers*, responsáveis por atender um dado pedido para um caminho, definidos no momento de criação do servidor. Foram abstraídos os elementos de pedido e resposta do HTTP para um fácil uso e interação. E construidos ajudantes para lidar com corpo de pedidos, em particular de formulários. Auxiliando assim o uso destes.

A API HTTP foi um dos componentes mais simples, sendo implementada sobre a ideia de

handlers definida no servidor Web, com cuidados para manter estado e vários componentes para ajudar a criar e lidar com os *plugins* e a biblioteca *multicast*. É ainda referido os objetos que a API lê e envia para o *browser* em formato JSON.

Na secção 4.5 é mostrado a aplicação base criada. Esta aplicação forma uma base em que se adiciona a aplicação Web desenvolvida, podendo ser compilada e distribuída. São mostrados os vários cuidados necessários para correr neste ambiente e adaptações que foram feitas em relação ao desenvolvido.

Capítulo 5

Aplicação exemplo

Para demonstrar o uso da biblioteca foi desenvolvida uma aplicação Web exemplo. Esta aplicação é criada para correr no *browser* e usa a biblioteca para partilhar dados com outros utilizadores. A aplicação demonstra:

- Criar uma nova sessão;
- Procurar uma sessão existente e ligar-se a esta;
- Criar um canal;
- Listar canais existentes;
- Enviar e receber de um canal;

A aplicação exemplo corre num navegador Web em Android e Windows, demonstrando assim grande parte da funcionalidade disponibilizada.

5.0.1 Interação com API HTTP

Para ajudar a interagir com a API HTTP foi desenvolvido um conjunto de *scripts* Javascript para fazer os pedidos e obter resposta destes. É usada a API XMLHttpRequest, seguindo um modelo semelhante ao apresentado no bloco de código 5.1. Sendo criada uma função para cada interação diferente na API, ajustando o método de acesso, os parâmetros e o caminho em que é feito o pedido.

```
function base(params, callback){
   var request=new XMLHttpRequest();
   httpRequest.onreadystatechange = function(e) {
        if (httpRequest.readyState === 4 && httpRequest.status === 200) {
            callback(httpRequest.responseText);
        }
    };
    httpRequest.open('POST', '/API/path' , true);// POST | GET | PUT
    httpRequest.send(data);
}
```

Bloco de Código 5.1: Javascript base para interação com API HTTP

5.0.2 Estrutura

A aplicação foi estruturada de forma a poder funcionar em Android e Windows. A sua estrutura está apresentada na figura 5.1 e é composta pelos ficheiros index.html o ponto de entrada da aplicação, um ficheiro de estilos CSS de nome style.css, o ficheiro helpers.js ajudante na interação com a API. Foi ainda usada a biblioteca de ícones Font Awesome [87] para construir botões mais apelativos visualmente e interativos. Além destes componentes foi adicionado também um dicionário de palavras necessário. Este dicionário foi obtido em [88] dado a sua licença (GNU LGPL v2.1) e ter mais de 2^{16} palavras.

Figura 5.1: Estrutura de ficheiros da aplicação Web

A aplicação não guarda estado, e o seu ciclo de vida e interação está modelado no diagrama

da figura 5.2.

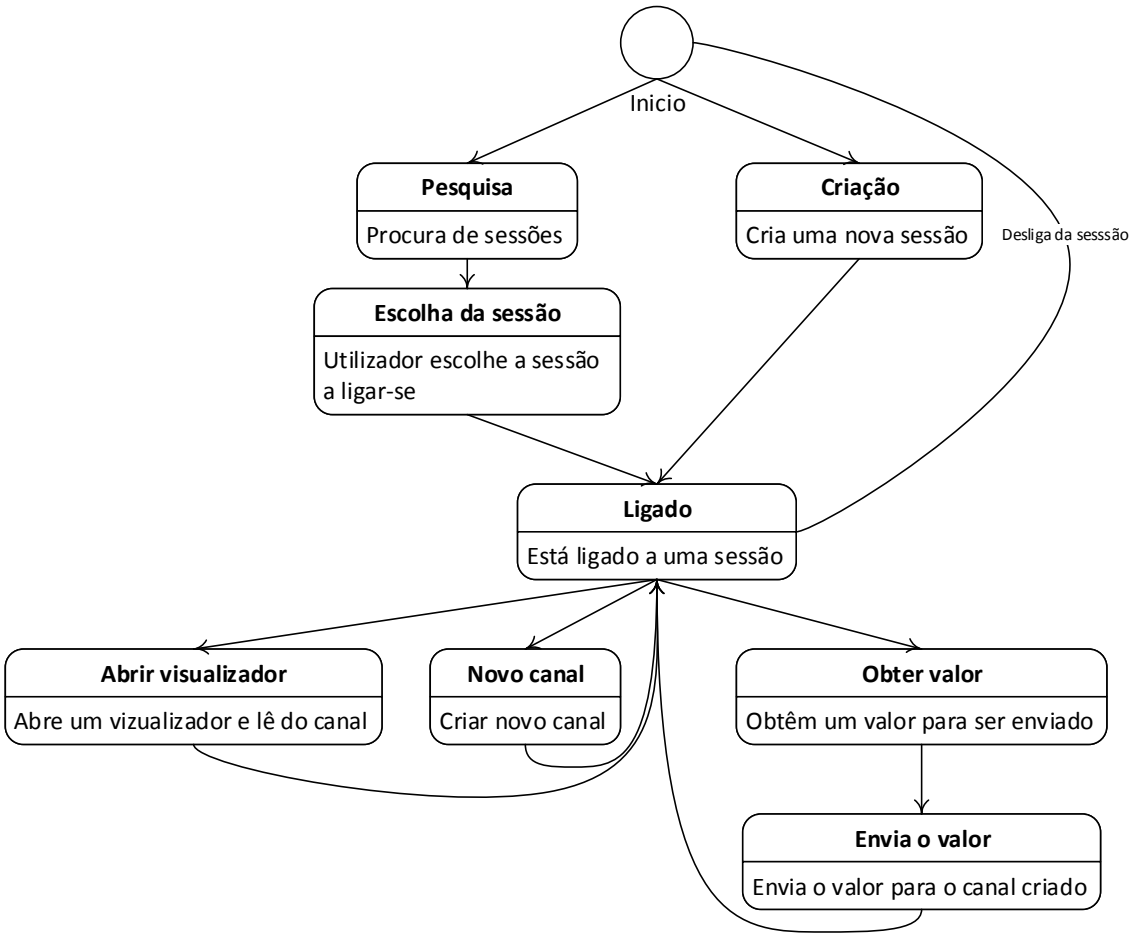


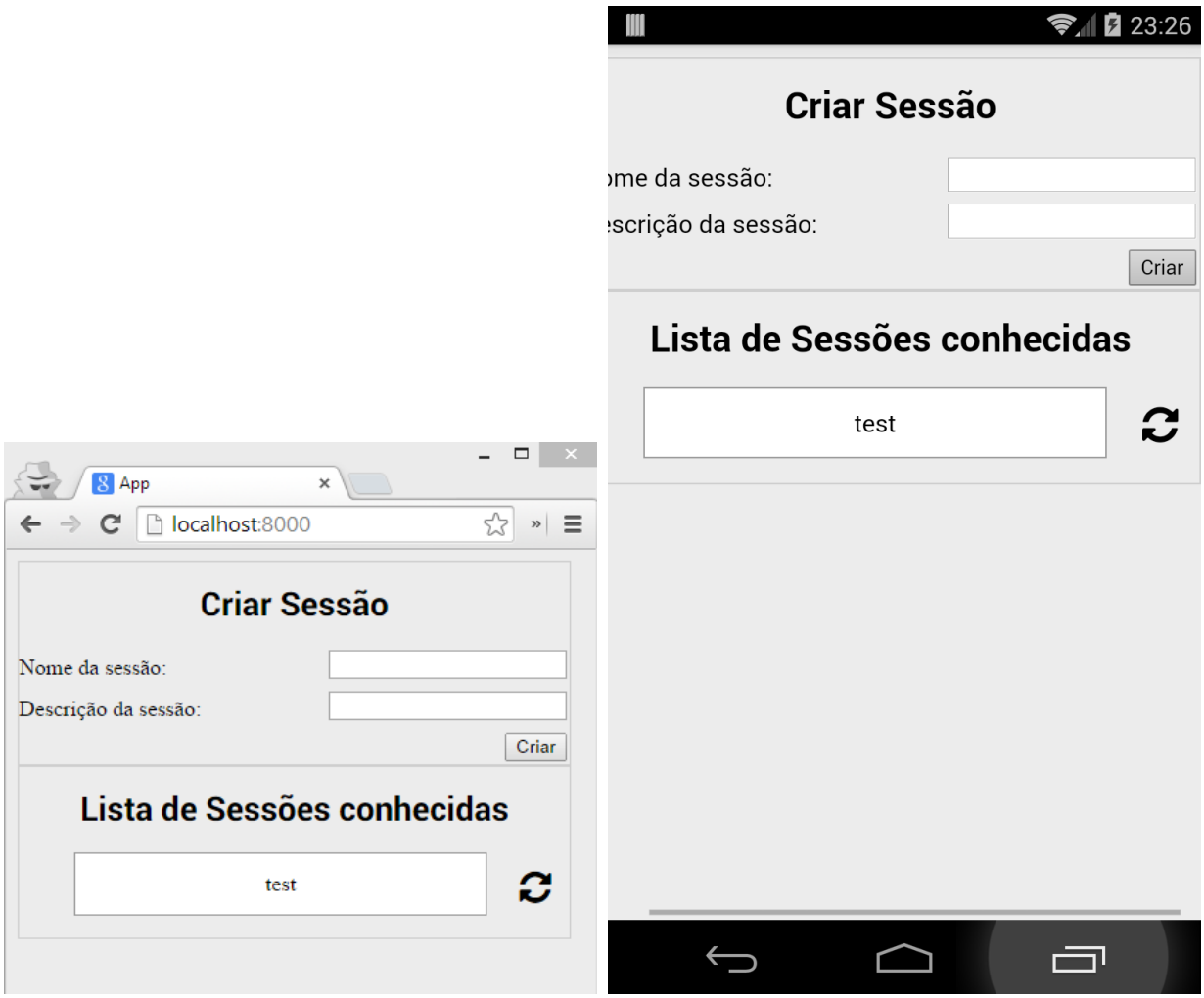
Figura 5.2: Interação com a aplicação desenvolvida

O utilizador começa por criar ou ligar-se a uma sessão, como pode ser visto na imagem 5.3. O utilizador para criar uma nova sessão simplesmente preenche o nome e a descrição e faz um pedido à API para criar uma nova sessão com aqueles parâmetros, enviando estes no objeto PublicSessionInfo definido na figura 4.10 e construído conforme mostra o bloco de código 5.2.

```
var session={name:name, description: description};
```

Bloco de Código 5.2: Javascript para criação do objeto a enviar para a criação de uma nova sessão

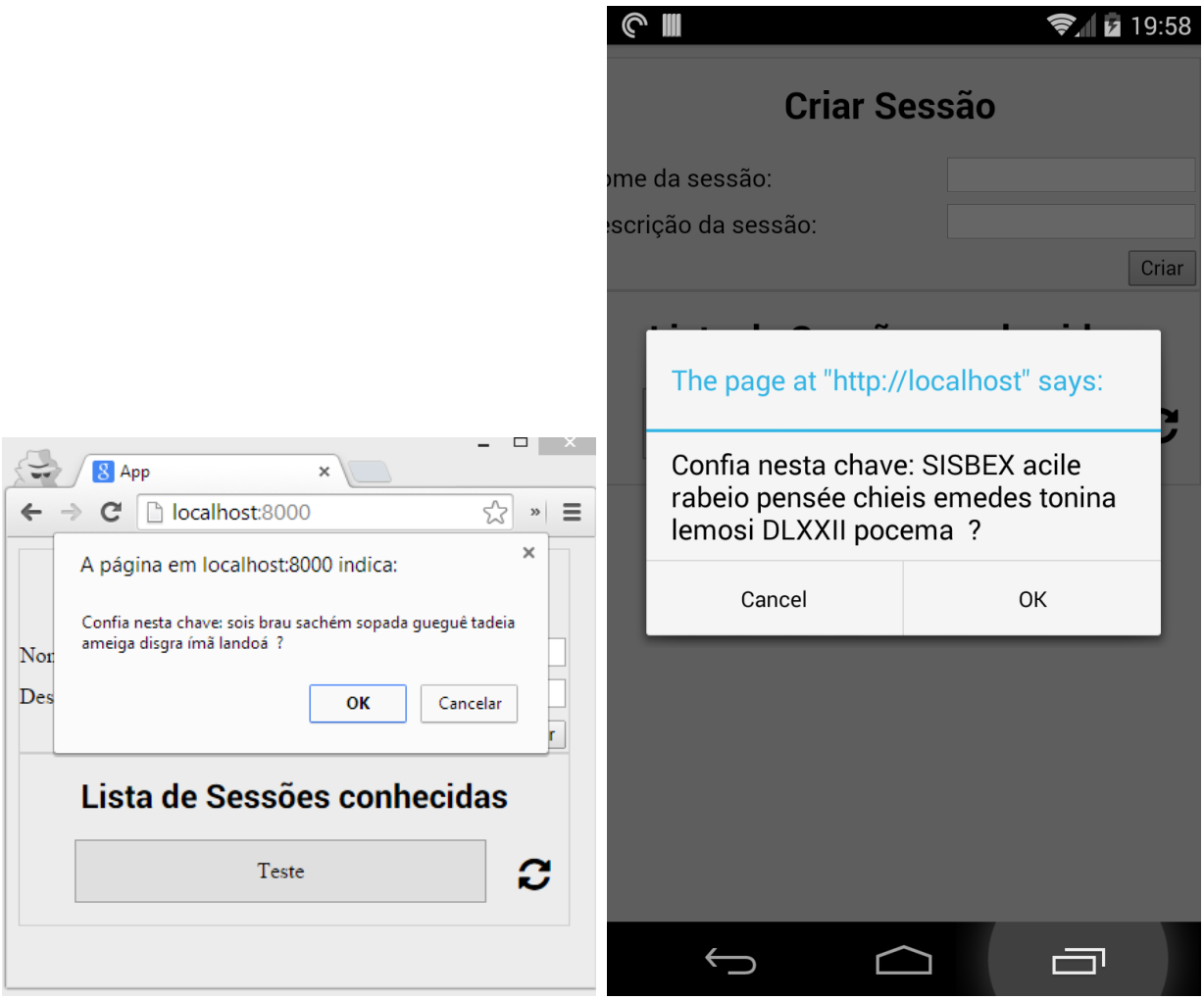
Para ligar-se a uma sessão o utilizador escolhe a sessão (clicando nesta) e é apresentado um *popup* para ele confirmar a *fingerprint* da chave pública (figura 5.4), caso confie nesta é lhe perguntado o seu nome e a palavra-chave de acesso à sessão (figuras 5.5 e 5.6 respetivamente).



(a) Aplicação em Chrome em Windows

(b) Aplicação em Android

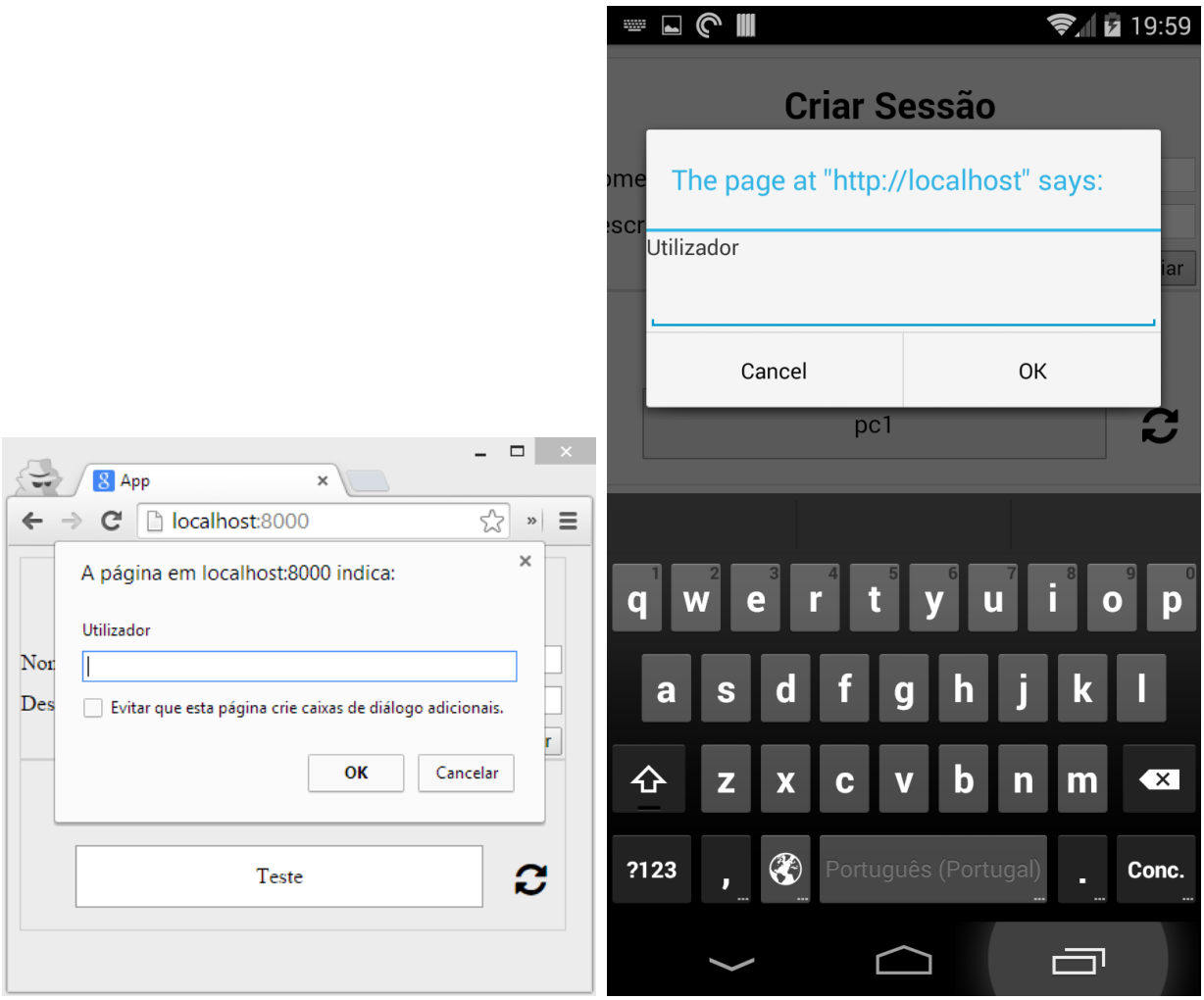
Figura 5.3: Vista para criar ou ligar-se a uma sessão



(a) Aplicação em Chrome em Windows

(b) Aplicação em Android

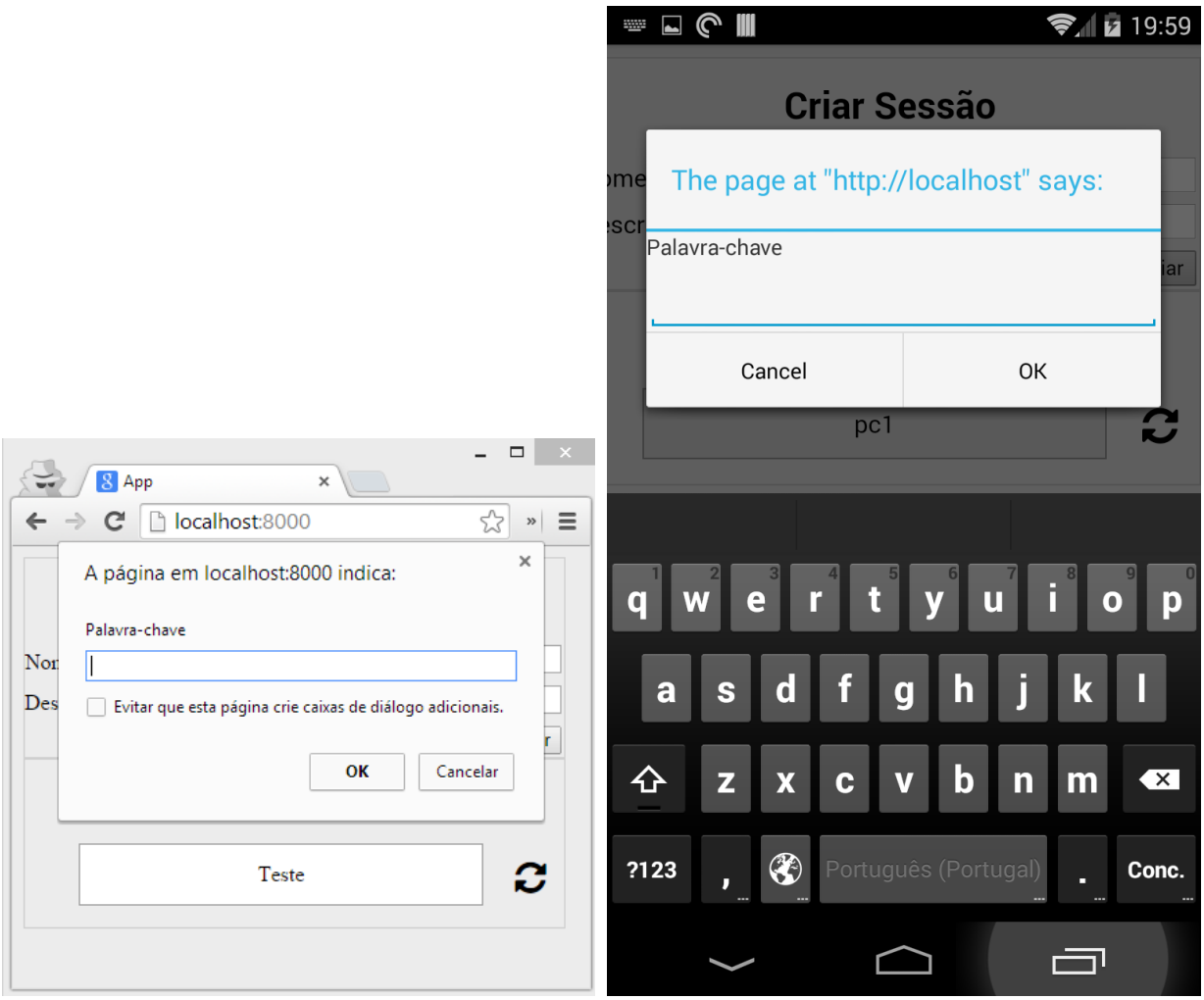
Figura 5.4: Confirmação da chave pública pelo utilizador na aplicação



(a) Aplicação em Chrome em Windows

(b) Aplicação em Android

Figura 5.5: Entrada do nome para ligar à sessão



(a) Aplicação em Chrome em Windows

(b) Aplicação em Android

Figura 5.6: Entrada da password para autenticar-se à sessão

(a) Aplicação em Chrome em Windows

(b) Aplicação em Android

Figura 5.7: Visualização de dados pelo visualizador

Capítulo 6

Conclusões

Este trabalho partiu com um conjunto de objetivos para ajudar a partilha de informações no ensino médico. A solução criada preenche todos os objetivos definidos na secção 1.2. Mas, fica a cargo de quem use este trabalho, criar uma aplicação que tome partido das possibilidades.

A aplicação Web exemplo forma um bom complemento ao trabalho, ilustrando o uso da API e do serviço. Mostra ainda que é possível ter a mesma aplicação a correr em várias plataformas. Este exemplo deve ser tido em conta para aplicações futuras que sejam criadas a usar este trabalho.

6.1 Trabalho Futuro

Devido a constrangimentos de tempo houve vários melhoramentos que não foram realizados e estudos que ficaram em falta.

Um dos mais importantes estudos a realizar seria uma avaliação da solução criada para fiabilidade em *multicast*. Em particular, seria interessante estudar o número de pacotes transmitidos na rede em comparação com uma solução baseada em conexões *unicast*, qual o tempo médio até todos os clientes receberem todos os pacotes. Seria ainda interessante estudar qual o valor ideal para passar a ignorar o pacote perdido na solução para tempo-real.

Relativamente a melhoramentos ao trabalho, a segmentação de mensagens seria uma adição útil. Isto permitiria que, um bloco de dados a ser enviado pela biblioteca pudesse ser segmentado em múltiplos pacotes de rede de forma a cada um caber num pacote UDP. No caso da solução

para tempo-real, um melhoramento necessário seria o de receber por completo cada bloco de dados ou descartá-lo. Em relação à recuperação de pacotes atual, também necessitava de melhoramentos para priorizar o envio de pacotes de forma a maximizar os fragmentos completos no lado do recetor.

Outro melhoramento na biblioteca *multicast* seria a criação de um algoritmo para ajustar dinamicamente o tempo de espera entre cada reenvio de pacotes no canal de controlo de forma a reduzir a sobrecarga na rede. Ajustando o tempo para um valor menor no evento de NACKs ou maior no caso de ninguém reportar perdas.

Para os canais de dados genéricos outro melhoramento discutido nos protocolos na secção 2.3 seria a possibilidade do uso de FEC, com ajuste dinâmico por canal tendo em conta o número de perdas detetadas.

A nível mais a baixo na *stack* protocolar outro melhoramento necessário serio o de ajuste do débito de envio dos pacotes para a rede, podendo ser ajustada por uma camada acima (a de Fiabilidade) para diminuir o débito caso existam muitas perdas. Atualmente isto não é feito porque espera-se que o tempo que demora a processar o pacote nas várias camadas até ser serializado e enviado seja o suficiente para criar um instante de espera entre pacotes, de forma a não sobrecarregar a rede.

Há vários aspetos da API fornecida ao browser que também poderiam ser melhorados. Um dos pontos em falta na API em relação ao existente na biblioteca *multicast* é a possibilidade de listar utilizadores. Falta também a implementação da API de notificações, o que torna a biblioteca complicada de usar (não é possível autorizar ou não na aplicação Web a criação de novos canais).

Sobre esta API de notificações deve ser também considerada aquando da sua implementação a adição de novos eventos, por exemplo, para permitir ao gestor aceitar ou rejeitar na sessão um utilizador já autenticado. Outra adição no processo de autenticação seria a de autenticação através de uma chave pública previamente conhecida pelo gestor da sessão.

Um último melhoramento à API seria a possibilidade da criação de canais para entrega em tempo-real. Esta funcionalidade já está implementado na biblioteca *multicast*, faltando apenas expô-la devidamente nos objetos para criar o canal com entrega em tempo real e nas informações dos canais existentes dar a informação do tipo de fiabilidade.

A segurança também podia ser melhorada, com o uso de um modo de cifragem que autentique

os dados cifrados [89]. Isto é, os dados cifrados neste modo para além de protegidos permitem detetar alterações que possam ter sofrido. Deveria ser também melhorado os servidores TCP e corrigido o problema apontado em 4.1.4.1 de poder abrir conexões ao servidor sem limites.

Para além dos melhoramentos já referidos, existem ainda algumas otimizações que poderiam fazer-se ao software. Em Android é reconhecido como problemático a criação de objetos, como tal deve ser minimizado o uso de **Enums** e outros objetos quando possível. Isto foi tido em conta em alguns locais em que foram usados **ints** onde **Enum** seria melhor opção, mas ainda seria possível melhorar. Mais informações sobre estes problemas podem ser encontrados em [90, 91].

A fim de minimizar o uso de memória, também deviam ser implementadas filas de armazenagem em disco e não em memória RAM quando possível. Um caso notável a destacar em que este melhoramento seria necessário é a fila que guarda os pacotes enviados/recebidos no módulo de **Fiabilidade**.

Bibliografia

[1] Rosalind Thomas. *Oral tradition and written record in classical Athens*. Number 18. Cambridge University Press, 1989.

[2] Doug Brent. Oral knowledge, typographic knowledge, electronic knowledge: Speculations on the history of ownership. *EJournal [serial online], I (3)*, 1991.

[3] Encyclopædia Britannica. optical scanner. http://www.britannica.com/EBchecked/topic/283266/optical-scanner, 2014. (Visitado em 03/09/2014).

[4] Visible Body. Visible Body | 3D Human Anatomy. http://www.visiblebody.com/index.html, 2014. (Visitado em 03/09/2014).

[5] Martin Dougiamas and Peter Taylor. Moodle: Using learning communities to create an open source course management system. In *World conference on educational multimedia, hypermedia and telecommunications*, volume 2003, pages 171–178, 2003.

[6] J.D. Trigo, A. Alesanco, I. Martínez, and J. García. A review on digital ECG formats and the relationships between them. *Information Technology in Biomedicine, IEEE Transactions on*, 16(3):432–444, May 2012. ISSN 1089-7771. doi:10.1109/TITB.2011.2176955.

[7] International Data Corporation (IDC). Worldwide smartphone shipments edge past 300 million units in the second quarter; android and ios devices account for 96according to idc - prus25037214. http://www.idc.com/getdoc.jsp?containerId=prUS25037214, Agosto 2014. (Visitado em 03/09/2014).

[8] StatCounter. Top 8 operating systems on aug 2014. http://gs.statcounter.com/#all-os-ww-monthly-201408-201408-bar, Agosto 2014. (Visitado em 03/09/2014).

[9] Google. Dashboards | android developers. https://developer.android.com/about/dashboards/index.html, Agosto 2014. (Visitado em 03/09/2014).

[10] Google. Android 4.4 kitkat and updated developer tools | android developers blog. http://android-developers.blogspot.pt/2013/10/android-44-kitkat-and-updated-developer.html, Outubro 2013. (Visitado em 03/09/2014).

[11] Google. Android, the world's most popular mobile platform | android developers. https://developer.android.com/about/index.html. (Visitado em 29/08/2014).

[12] Mike Cleron. Using webviews | android developers blog. http://android-developers.blogspot.pt/2008/09/using-webviews.html, Setembro 2008. (Visitado em 29/08/2014).

[13] Apache. Apache Cordova. https://cordova.apache.org/. (Visitado em 29/08/2014).

[14] Adobe. Phonegap | about. http://phonegap.com/about/. (Visitado em 29/08/2014).

[15] Brian LeRoux. Phonegap | phonegap, cordova, and what's in a name? http://phonegap.com/2012/03/19/phonegap-cordova-and-what%E2%80%99s-in-a-name/, Março 2012. (Visitado em 29/08/2014).

[16] Google. The Chromium Projects. http://www.chromium.org/Home. (Visitado em 23/09/2014).

[17] Google. Chromium Blog: Run Chrome Apps on mobile using Apache Cordova. http://blog.chromium.org/2014/01/run-chrome-apps-on-mobile-using-apache.html, Janeiro 2014. (Visitado em 04/09/2014).

[18] Juhani Lehtimäki. Android UI Patterns: Multi-platform Frameworks Destroy Android UX. http://www.androiduipatterns.com/2012/03/multi-platform-frameworks-destroy.html, Março 2012. (Visitado em 04/09/2014).

[19] Google. Introduction - Material Design - Google design guidelines. http://www.google.com/design/spec/material-design/introduction.html, 2014. (Visitado em 04/09/2014).

[20] rogerwang. rogerwang/node-webkit. https://github.com/rogerwang/node-webkit, Agosto 2014. (Visitado em 29/08/2014).

[21] Ian S. Graham. *The HTML SourceBook*. John Wiley & Sons, Inc., New York, NY, USA, 1995. ISBN 0471118494.

[22] Håkon Wium Lie and Bert Bos. Cascading Style Sheets, level 1. *W3C, www. w3. org/pub/www/TR/REC-CSS1-961217*, pages 1–86, 1996.

[23] Gordon McComb. *JavaScript sourcebook: create interactive JavaScript programs for the World Wide Web*. John Wiley & Sons, Inc., 1996.

[24] B. Lawson and R. Sharp. *Introducing HTML5*. Pearson Education, 2011. ISBN 9780132792974.

[25] W3C. XMLHttpRequest Level 1. http://www.w3.org/TR/XMLHttpRequest/. (Visitado em 29/08/2014).

[26] Dave Crane and Phil McCarthy. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Apresspod Series. Apress, 2008. ISBN 9781590599983.

[27] Exploring reverse ajax. http://gmapsdotnetcontrol.blogspot.pt/2006/08/exploring-reverse-ajax-ajax.html, Agosto 2006. (Visitado em 29/08/2014).

[28] W3C. Server-sent events. http://www.w3.org/TR/eventsource/, Dezembro 2012. (Visitado em 29/08/2014).

[29] Google. WebRTC. http://www.webrtc.org/. (Visitado em 29/08/2014).

[30] W3C. WebRTC 1.0: Real-time Communication Between Browsers. http://www.w3.org/TR/webrtc/, Setembro 2013. (Visitado em 29/08/2014).

[31] IETF. Rtcweb status pages. http://tools.ietf.org/wg/rtcweb/. (Visitado em 29/08/2014).

[32] WebRTC Audio Codec and Processing Requirements. http://tools.ietf.org/html/draft-ietf-rtcweb-audio-05, Fevereiro 2014. (Visitado em 29/08/2014).

[33] IETF. WebRTC Video Processing and Codec Requirements. http://tools.ietf.org/html/draft-ietf-rtcweb-video-00, Julho 2014. (Visitado em 29/08/2014).

[34] Irwin Lazar. WebRTC Video Codec Unresolved - Post - No Jitter. http://www.nojitter.com/post/240164030/webrtc-video-codec-unresolved, Novembro 2013. (Visitado em 29/08/2014).

[35] Jonathan Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245, IETF, April 2010.

[36] Eric Rescorla. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, IETF, April 2010.

[37] Philipp Hancke. Is WebRTC ready yet? http://iswebrtcreadyyet.com/, Julho 2014. (Visitado em 29/08/2014).

[38] Jason Kersey. Chrome Releases: Chrome for Android Update. http://googlechromereleases.blogspot.pt/2013/08/chrome-for-android-update.html, Agosto 2013. (Visitado em 29/08/2014).

[39] Twilio. Which browsers support WebRTC? https://www.twilio.com/help/faq/twilio-client/which-browsers-support-webrtc. (Visitado em 29/08/2014).

[40] StatCounter. Top 9 browsers from july 2013 to july 2014 | statcounter global stats. http://gs.statcounter.com/#all-browser-ww-monthly-201307-201407, Agosto 2014. (Visitado em 29/08/2014).

[41] Microsoft. Internet Explorer Web Platform Status and Roadmap - WebRTC. http://status.modern.ie/webrtcwebrtcv10api. (Visitado em 29/08/2014).

[42] Google. WebView for Android - Google Chrome. https://developer.chrome.com/multidevice/webview/overview, 2014. (Visitado em 29/08/2014).

[43] Ian Fette and Alexey Melnikov. The WebSocket Protocol. RFC 6455, IETF, December 2011.

[44] Hubert Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, 1980.

[45] Dah Ming Chiu, Stephen Hurst, Miriam Kadansky, and Joseph Wesley. Tram: A tree-based reliable multicast protocol. 1998.

[46] Tie Liao. Light-weight reliable multicast protocol. 1998.

[47] Michael Luby, Lorenzo Vicisano, Jim Gemmell, Luigi Rizzo, M Handley, and Jon Crowcroft. The use of forward error correction (FEC) in reliable multicast. RFC 3453, IETF, December 2002.

[48] Jim Gemmell, Todd Montgomery, Tony Speakman, Nidhi Bhaskar, and Jon Crowcroft. The PGM Reliable Multicast Protocol. Institute of Electrical and Electronics Engineers, Inc., March 2003.

[49] Jonas Ådahl. jadahl/jgroups-android. https://github.com/jadahl/jgroups-android, 2009. (Visitado em 29/08/2014).

[50] Red Hat. JGroups - Overview. http://www.jgroups.org/overview.html. (Visitado em 29/08/2014).

[51] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[52] J. Rosenberg and H. Schulzrinne. An offer/answer model with the session description protocol (SDP). RFC 3264, IETF, June 2002.

[53] S. Nandakumar and C. Jennings. SDP for the WebRTC. Draft draft-nandakumar-rtcweb-sdp-05, IETF, August 2014.

[54] David McGrew and Eric Rescorla. Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP). Draft 5764, IETF, May 2010.

[55] E. Rescorla. WebRTC Security Architecture. http://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-10, Julho 2014. (Visitado em 29/08/2014).

[56] Jason Fischl, Eric Rescorla, and Hannes Tschofenig. Framework for establishing a secure real-time transport protocol (SRTP) security context using datagram transport layer security (DTLS). *Framework*, 2010.

[57] Eric Rescorla. Diffie-hellman key agreement method. RFC 2631, IETF, June 1999.

[58] Mark Handley and V Jacobson. SDP: Session description protocol. RFC 2327, IETF, April 1998.

[59] C. Jennings. SDP for the WebRTC. http://tools.ietf.org/id/draft-nandakumar-rtcweb-sdp-01.html, Fevereiro 2013. (Visitado em 29/08/2014).

[60] Van Jacobson, Ron Frederick, Steve Casner, and H Schulzrinne. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, IETF, July 2003.

[61] Sebastiano Gottardo. [dex] sky's the limit? no, 65k methods is — medium. https://medium.com/@rotxed/dex-skys-the-limit-no-65k-methods-is-28e6cb40cf71, Julho 2014. (Visitado em 05/09/2014).

[62] C. Adams and S. Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Technology series. Addison-Wesley, 2003. ISBN 9780672323911.

[63] Donald Eastlake and Paul Jones. Us secure hash algorithm 1 (sha1). RFC 3174, IETF, September 2001.

[64] Mozilla. Ca:problematic practices - sha-1 certificates. https://wiki.mozilla.org/CA:Problematic_Practices#SHA-1_Certificates, Agosto 2014. (Visitado em 09/09/2014).

[65] Eric Mill. Why google is hurrying the web to kill sha-1. https://konklone.com/post/why-google-is-hurrying-the-web-to-kill-sha-1, Setenbro 2014. (Visitado em 09/09/2014).

[66] Oracle. Securerandom (java platform se 7). http://docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html. (Visitado em 23/09/2014).

[67] Morris Dworkin. Recommendation for block cipher modes of operation. methods and techniques. Technical report, DTIC Document, 2001.

[68] John Chanak and William Kish. Throughput enhancement by acknowledgement suppression, Novembro 4 2005. US Patent App. 11/267,477.

[69] Alexis Deveria. Can i use... support tables for html5, css3, etc. http://caniuse.com/#feat=xhr2, Julho 2014. (Visitado em 30/08/2014).

[70] Douglas Crockford. JSON: The fat-free alternative to XML. In *Proc. of XML*, volume 2006, 2006.

[71] Microsoft. Getsockopt() function returns a different maximum udp message size than you expect in windows 2000 sp3. https://support.microsoft.com/kb/822061. (Visitado em 30/08/2014).

[72] G.Bianchi, G.Neglia, and V.Mancuso. Internet transport layer: User datagram protocol (udp). http://www-sop.inria.fr/members/Vincenzo.Mancuso/ReteInternet/05_udp.pdf. (Visitado em 30/08/2014).

[73] Google. Protocol buffers - google's data interchange format - google project hosting. https://code.google.com/p/protobuf/. (Visitado em 30/08/2014).

[74] Igor Anishchenko. Thrift vs protocol buffers vs avro - biased comparison. http://www.slideshare.net/IgorAnishchenko/pb-vs-thrift-vs-avro, Setembro 2012. (Visitado em 30/08/2014).

[75] David Yu. Benchmarkingv2 - thrift-protobuf-compare - comparing varius aspects of serialization libraries on the jvm platform - google project hosting. https://code.google.com/p/thrift-protobuf-compare/wiki/BenchmarkingV2, Janeiro 2011. (Visitado em 30/08/2014).

[76] Martin Kleppmann. Schema evolution in avro, protocol buffers and thrift — martin kleppmann's blog. http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html, Dezembro 2012. (Visitado em 30/08/2014).

[77] Apache. Apache Thrift - Home. https://thrift.apache.org/, 2014. (Visitado em 30/08/2014).

[78] Apache. Welcome to Apache Avro! https://avro.apache.org/, Julho 2014. (Visitado em 30/08/2014).

[79] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–HTTP/1.1. RFC 2616, IETF, June 1999.

[80] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. Hypertext transfer protocol–HTTP/1.0. RFC 1945, IETF, May 1996.

[81] World Wide Web Consortium et al. Html 4.01 specification. 1999.

[82] Google. google-gson - a java library to convert json to java objects and vice-versa - google project hosting. https://code.google.com/p/google-gson/. (Visitado em 30/08/2014).

[83] Apache. Apache License, Version 2.0. https://www.apache.org/licenses/LICENSE-2.0, Janeiro 2014. (Visitado em 30/08/2014).

[84] Google. Gson on android - gson. https://sites.google.com/site/gson/gson-on-android. (Visitado em 16/09/2014).

[85] Brian W Kernighan. The unix system and software reusability. *Software Engineering, IEEE Transactions on*, (5):513–518, 1984.

[86] Google. Application fundamentals | android developers. https://developer.android.com/guide/components/fundamentals.html. (Visitado em 16/09/2014).

[87] Dave Gandy. Font awesome, the iconic font and css toolkit. http://fontawesome.io/, 2014. (Visitado em 17/09/2014).

[88] WinEdt. Winedt.org – dictionaries. http://www.winedt.org/Dict/, Março 2012. (Visitado em 18/09/2014).

[89] Petr Švenda. Basic comparison of Modes for Authenticated-Encryption (IAPM, XCBC, OCB, CCM, EAX, CWC, GCM, PCFB, CS).

[90] Chet Haase and Romain Guy. Part 1: Android performance workshop. http://www.parleys.com/play/5298f999e4b039ad2298c9e3/chapter57/about, Dezembro 2013. (Visitado em 17/09/2014).

[91] Google. Performance tips | android developers. https://developer.android.com/training/articles/perf-tips.html#avoid_enums. (Visitado em 17/09/2014).

Apêndice A

Acrónimos

ACK Acknowledge

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

CMS Content Management System

CSS Cascading Style Sheets

DLRs Designated Local Repairers

DLR Designated Local Repairer

DOM Document Object Model

DTLS Datagram Transport Layer Security

FEC Forward Error Correction

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

ICE Interactive Connectivity Establishment

IETF Internet Engineering Task Force

JSON JavaScript Object Notation

LRMP Light-weight Reliable Multicast Protocol

NACK Not Acknowledge

NCF NAK confirmation

NE Network Elements

OSI Open Systems Interconnection

PGM Pragmatic General Multicast

REST Representational State Transfer

RFC Request for Comments

RTP Real-time Transport Protocol

SDK Software Developement Kit

SDP Session Description Protocol

SSE Server-Sent Events

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TRAM Tree-Based Reliable Multicast

TTL Time To Live

TURN Traversal Using Relay NAT

UDP User Datagram Protocol

UML Unified Modeling Language

URL Uniform Resource Locator

VM máquina virtual

W3C World Wide Web Consortium

XML Extensible Markup Language